异步图书
www.epubit.com.cn

龚鹏 著

微服务

分布式构架开发实战

Spring Boot Spring Cloud
Spring Data Swagger2
Nexus Jenkins
Dubbo ELK
QuartZ

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

龚鹏

全栈GEEK，高中时期开始自学Java，痴迷互联网技术，曾服务奥美互动、百度、中青旅，在车联网设计、互联网彩票、电子商务系统开发方面积累了丰富的经验。

读者QQ交流群：259280854

微服务

分布式构架开发实战

龚鹏 著

人民邮电出版社
北京

图书在版编目（CIP）数据

微服务分布式构架开发实战 / 龚鹏著. -- 北京 : 人民邮电出版社, 2018.2
ISBN 978-7-115-47558-9

Ⅰ. ①微… Ⅱ. ①龚… Ⅲ. ①分布控制－控制系统－系统设计 Ⅳ. ①TP273

中国版本图书馆CIP数据核字(2017)第318708号

内 容 提 要

随着第三方框架的逐渐完善，实施微服务架构的开发成本越来越低，分布式架构成为主流势不可挡。一个完善的架构或系统中包含了许多的知识点，而每一个知识点则又可以引出非常多的内容，过度地专注于细节反而会拖慢达成目标的步伐。为了更快地实施微服务，本书基于开源且稳定的第三方工具，介绍如何构建一个庞大且复杂的分布式系统，用于满足项目中的实际需求。

每一个工具库为了适应更丰富的使用场景，通常都会把部分参数以配置文件的方式暴露出来，同时提供用于开发环境的默认配置。本书基于快速使用为主线，尽可能多地讲解配置参数的意义及它们之间的关系，帮助读者在掌握足够多的知识点后，建立起对微服务分布式架构的认知，以便为探求更深层次的知识点做好铺垫。

本书适合 Java 工程师、初级架构师、大中专院校相关专业师生、Java 培训班学员及独立开发者与自学读者使用。

◆ 著　　龚　鹏
责任编辑　赵　轩
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市君旺印务有限公司印刷

◆ 开本：800×1000　1/16
印张：13.5
字数：341 千字　　2018 年 2 月第 1 版
印数：1－3 000 册　　2018 年 2 月河北第 1 次印刷

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

序

龚鹏同学的这本书全面覆盖了微服务的细分领域，以一线实战视角详述了如何实现微服务。一方面，读者可以通过本书拓宽技术视野，另一方面，读者也可以从其中的一章深入了解一个细分领域的微服务实践。

在阿里巴巴的生态中，微服务逐渐成为主要的服务形态，伴随着容器化的日臻成熟，大量的分布式、领域驱动设计的微服务被快速开发和部署，服务间呈现出搭积木的能力，使不同的业务通过重新组合数个微服务，就能实现新的业务场景。借助成熟的底层集团中间件，天然地支持微服务所需的动态扩缩、服务发现、全链路日志分析等能力。以阿里巴巴达摩院语音对话平台为例，对话服务、理解服务、问答服务，以及对话管理平台都是基于 SpringBoot 和 Docker 技术栈的。

希望读者能够通过本书的指引，在自己的工作和学习领域，快速并逐渐深入地建立起自己的微服务。

韩陆

《Java RESTful Web Service 实战》作者

达摩院语音对话平台负责人

前言

PREFACE

在当下的互联网世界里，系统开发既简单又困难。说它简单，是因为各技术社区的贡献及各框架的完善，使整个生态越来越丰富，绝大多数的需求都可以使用现有的库实现（不用重新发明“轮子”），并且现在很多的库都非常注重使用体验，以快速实现为目标，通过少量的配置及代码便可集成使用。但合理地编排这些库以构建一个高效、稳健、灵活的系统，并不是一件容易的事，人们一般将专精此道的人称为架构师。

一个完善可靠的微服务分布式架构需要解决众多的问题，我们可以用多种方法去解决这些问题，但刚开始接触时很难确定哪种方法是最优方案。在不确定并且刚接触如此庞大的架构时，选择信任、成熟且活跃的框架及第三方库提供的解决方案显然是明智之举。当建立起对微服务分布式架构的认知后，再重新回归最初的问题逐步审视并深入，最终形成属于自己的方案。

以往的图书大多只针对微服务分布式架构自身的知识点讲解，周边的相关知识点并未涉及，在进行项目实践时，最终效果则根据读者自身的知识储备而定。

本书特色

从实际出发

本书并没有过多地探讨理论性的内容，而是基于现有成熟框架，围绕实际项目中遇见的具体需求，以微服务分布式架构的角度去逐一分解并且实现这些需求。掌握这些知识的读者，完全有能力快速搭建出可靠、高效、灵活的微服务分布式架构。

与行业动向接轨

借助于现有框架进行微服务分布式架构实践的成本越来越低，并且这种形式正在逐渐成为主流架构。在进行框架及第三方库的选择时，也同样紧跟行业动向。

简单易懂

本书中的每一个示例都尽量用最少的代码和最快的方式解决具体的问题，为读者呈现复杂系统中简单的一面，其目的是快速接受并理解各知识点在微服务分布式架构中所处的位置及其相互关系。

本书面向读者

- Java 工程师
- 初级架构师

- 独立开发者与自学读者
- 高校相关专业师生
- Java 培训班学员

学习前的一些建议

明确目标

技术服务于商业，尽量避免陷入技术细节的漩涡中，不要为了技术而技术。在进入学习状态之前，根据书中的章节与自己的经验明确学习目标，不断地提出问题并验证，最终找出答案。

先定个小目标

面对一个复杂架构体系，从简单的知识点入手逐个攻破，用小成就感驱动自己最终完成设定的计划。

持之以恒

学习从来都不是一件容易的事，从学习 ABC 到写出自己的第一篇文章，从建立账号到超越第一个 BOSS 或者赢得第一场胜利，这之间的过程都是学习，都需要投入大量的时间和精力。如果你想要做出心目中理想的网站，或者将其作为自己赖以谋生的技能，坚持下去，你才能够做到。

积极探索

互联网技术的优势是，当你遇到问题时，往往可以在互联网上寻求答案。互联网行业的大牛同样活跃在互联网上，找到他们，向他们学习。订阅公众号或者相关博客，积极了解行业发展和最新动向。这是其他学科无法比拟的优势。

勘误与联系方式

在本书写作出版过程中，无论是作者还是编辑，都希望本书能够尽善尽美。如果您发现了本书的不足或错漏之处，欢迎您不吝赐教，帮助我们改进提升本书的内容，您可以通过以下方式联系我们。

QQ 群

我们为本书专门提供了一个 QQ 交流群（群号 259280854），读者不仅可以在此向本书作者反馈建议，还能和其他读者共同交流，一起成长。

作者邮箱

如果您有任何问题，都可以直接与作者联系，电子邮件地址为 gongroc@outlook.com。我会尽快与您联系，解答您的疑问。

异步社区

在异步社区 http://www.epubit.com.cn/ 中搜索到本书页面，您便可以下载本书相关素材，还可以提交本书勘误。勘误被确认后，我们会向您提供积分奖励，这些积分可以在社区使用，如购书优惠、换领样书。

目 录

CONTENTS

第1章 微服务介绍 …… 1

1.1 什么是微服务架构 …… 2
1.2 垂直应用与微服务 …… 2
1.3 实现一个最简单的微服务框架 …… 3
1.3.1 公共接口 …… 4
1.3.2 服务端 …… 4
1.3.3 客户端 …… 7
1.3.4 完善框架 …… 8
1.4 主流微服务框架介绍 …… 9
1.4.1 Dubbo …… 9
1.4.2 Spring Cloud …… 10

第2章 模块拆分 …… 12

2.1 拆分逻辑 …… 13
2.2 单模块 …… 14
2.3 基础模块 …… 14
2.4 复杂模块 …… 15

第3章 Spring Boot …… 16

3.1 目录结构 …… 17
3.2 主要文件 …… 18
3.3 编辑器集成 …… 18

第4章 Dubbo …… 20

4.1 注册中心 …… 21
4.2 接口工程 …… 22
4.3 服务端 …… 23
4.4 消费方 …… 28
4.5 网关 …… 30
4.6 监控中心 …… 33
4.7 服务管理 …… 35
4.8 负载均衡 …… 36
4.9 服务降级 …… 37
4.10 集群容错 …… 38

第5章 Spring Cloud …… 40

5.1 注册中心 …… 41
5.2 注册服务 …… 44
5.3 调用服务 …… 45
5.3.1 Ribbon …… 46
5.3.2 Feign …… 49
5.4 Zuul网关 …… 51
5.5 Hystrix 断路器 …… 54
5.5.1 Ribbon …… 54
5.5.2 Fegin …… 57
5.6 服务监控 …… 58
5.7 应用监控 …… 61
5.8 熔断器监控 …… 62
5.8.1 单应用的熔断数据 …… 63
5.8.2 使用Turbine聚合数据 …… 64
5.8.3 Cloud Admin整合Turbine 65

5.9 统一管理配置文件 …… 66

第6章 数据持久化 …… 70

6.1 Spring Data MySQL …… 71
6.1.1 依赖与配置 …… 71
6.1.2 实体映射 …… 72
6.1.3 Repository …… 76
6.1.4 JdbcTemplate …… 79
6.1.5 事务管理 …… 80
6.2 Spring Data MongoDB …… 81
6.2.1 依赖与配置 …… 81
6.2.2 实体映射 …… 82
6.2.3 Repository …… 83
6.2.4 MongoTemplate …… 84
6.3 Spring Data ElasticSearch …… 85
6.3.1 基本概念 …… 85
6.3.2 安装与运行 …… 86
6.3.3 基于HTTP交互 …… 87
6.3.4 配置分词器 …… 91
6.3.5 依赖与配置 …… 94
6.3.6 实体映射 …… 94
6.3.7 Repository …… 95
6.3.8 ElasticsearchTemplate … 96
6.4 TCC 分布式事务 …… 98
6.5 Spring Data Redis …… 100
6.5.1 安装运行 …… 100
6.5.2 依赖与配置 …… 101
6.5.3 缓存支持 …… 102
6.5.4 RedisTemplate …… 106
6.5.5 全局锁 …… 107

第7章 表单验证 …… 110

第8章 定时任务 …… 113

8.1 Spring Task 单机定时任务 … 114
8.2 Cron 表达式 …… 114
8.3 QuartZ 分布式定时任务 …… 116

第9章 分布式会话 …… 122

第10章 消息队列 …… 124

10.1 安装及配置RabbitMQ …… 125
10.2 配置及使用 …… 128

第11章 构建Web应用 …… 130

第12章 异常处理 …… 133

第13章 安全认证 …… 139

13.1 OAuth2.0 协议介绍 …… 140
13.2 授权模式 …… 141
13.3 在Dubbo中使用OAuth 2.0 142
13.4 在Spring Cloud 中使用OAuth 2.0 …… 151
13.4.1 授权中心 …… 151
13.4.2 服务模块配置 …… 154
13.4.3 网关配置 …… 157
13.4.4 测试运行 …… 159

第14章 日志管理 …… 161

14.1 Spring Boot 日志 …… 162
14.1.1 日志格式 …… 162
14.1.2 输出到文件 …… 163
14.1.3 扩展配置 …… 163
14.2 分布式日志管理 …… 166

14.2.1 ELK 搭建 …… 167
14.2.2 Spring Boot 配置 …… 169

第15章 热部署 …… 171

第16章 接口文档管理 …… 173

16.1 Dubbo中使用Swagger2…… 174
16.2 Spring Cloud中使用 Swagger2 …… 178
16.2.1 微服务模块配置 …… 179
16.2.2 网关模块配置 ……181

第17章 Nexus私库 …… 184

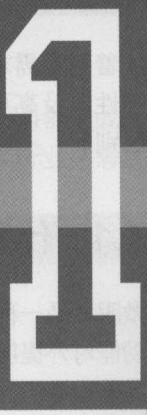
17.1 Nexus 安装 …… 185
17.2 从Nexus私库下载jar包…… 187
17.3 上传jar包到Nexus私库 …… 189

第18章 发布系统 …… 191

18.1 Jenkins 安装配置 …… 192
18.2 构建任务 …… 194

第19章 分布式架构总结 …… 201

第1章　微服务介绍



1.1　什么是微服务架构

1.2　垂直应用与微服务

1.3　实现一个最简单的微服务框架

1.4　主流微服务框架介绍

随着用户需求个性化、产品生命周期变短，微服务架构是未来软件架构朝着灵活性、扩展性、伸缩性以及高可用性发展的必然方向。这里主要将对比传统的垂直应用与分布式微服务应用之间的区别。

1.1　什么是微服务架构

微服务是一种软件架构风格，目标是将一个复杂的应用拆分成多个服务模块，每个模块专注单一业务功能对外提供服务，并可以独立编译及部署，同时各模块间互相通信彼此协作，组合为整体对外提供完整服务。

> 微服务架构就像是活字印刷术，每个文字模都可以看成是一个微服务，它可以独立地提供印刷服务，又可以将模块之间组合，最终形成一篇完整文章提供更为复杂的印刷服务。

由于每个模块都独立部署，各自拥有互不干扰的内存空间，模块之间无法直接调用，所以需要借助 RPC（远程过程调用协议）或 HTTP 协议让各个模块之间传递通信报文及交换数据，实现远程调用，整个通信管理的过程也是微服务架构重要的组成部分。

1.2　垂直应用与微服务

MVC 模式构建的垂直应用非常适合项目初期，使用其能够方便地进行开发、部署、测试，但随着业务的发展与访问量的增加，垂直应用的问题也随之暴露出来，而微服务架构可以很好地解决这些问题。

代码维护

垂直应用里，大部分逻辑都部署在一个集中化、单一的环境或服务器中运行。垂直应用程序通常很大，由一个大型团队或多个团队维护。庞大的代码库可能给希望熟悉代码的开发人员增加学习成本，还会让应用程序开发过程中使用的开发环境工具和运行容器不堪重负，最终导致开发效率降低，可能会阻止对执行更改的尝试。

微服务架构将这个庞大并且复杂的应用拆分成多个逻辑简单且独立的小应用，每个小应用交由不同的团队或开发人员维护，彼此之间互不干扰，通过标准接口互相通信。对于希望熟悉代码的开发人员来说只需掌握他所负责的应用即可，这样做的好处是简单、快速、逻辑清晰。

部署

垂直应用需要处理一个庞大的应用程序，编译、部署需要花费很长时间，一个小的修改就可能导致重新构建整个项目。

微服务架构中对其中某一个服务进行修改，只需重新编译、部署被改动的服务模块。

资源控制

垂直应用里，当请求量过大导致单台服务器无法支撑时，一般会将垂直应用部署在多台服务器形成服务集群，并通过反向代理实现负载均衡。集群中的每个服务必须部署完整的应用，但在实际业务需求中仅有部分功能使用频繁，但这种架构必须为不常用的功能分配计算资源。

微服务将提供功能的各服务拆分为多个服务模块，它具有天生的集群属性，能够轻松地根据用量部署。

例如系统中的消息功能使用频率占了整个系统的 90%，而密码找回功能则只占到 2%。为了分解消息功能的压力，以传统负载均衡的方式进行集群化时，每个服务必须为使用量只有 2% 的密码找回功能分配资源，这无疑造成了浪费。

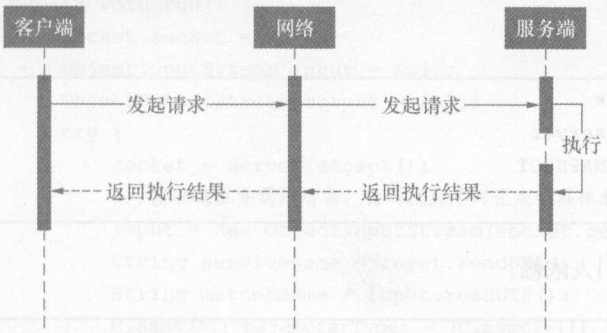
在微服务架构中，消息功能使用率占据 90%，则将消息模块多部署几个实例形成集群，而密码找回功能所在的用户模块只部署一个就可以了。

稳定

垂直应用中如果有一个小的问题，就可能使整个系统崩溃。

微服务所拆分出的各个模块中，由于模块之间的耦合度很低，当发生问题时影响范围被固定在该模块本身，整个系统依然健全。

1.3 实现一个最简单的微服务框架



基本工作流程如下。

① 客户端发起调用请求。

② 将调用的内容序列化后通过网络发给服务端。

③ 服务端接收到调用请求，执行具体服务并获得结果。

④ 将结果序列化后通过网络返回给客户端。

1.3.1　公共接口

在发起远程调用时，需要基于接口（Interface）来约定客户端与服务端所调用服务的具体内容。为了方便管理依赖关系，这里使用 Maven 构建应用并编写一些接口，以提供给客户端与服务端使用。

当然也可以使用普通的 Java 应用来实现此简单微服务框架，只需将该应用编译后的 jar 包提供给后续的服务端与客户端即可。

Maven 参数

```
groupId: org.book
artifactId: rpc-interface
version: 0.0.1-SNAPSHOT
packaging: jar
```

编写接口。

```
public interface HelloService {
    public String hello(String name);
}
```

1.3.2　服务端

新建用于提供服务的 Maven 应用，并引入刚编写的接口应用依赖。

Maven 参数

```
groupId: org.book
artifactId: rpc-server
version: 0.0.1-SNAPSHOT
packaging: jar
```

① 在 pom.xml 文件中引入依赖。

```
<dependency>
    <groupId>org.book</groupId>
    <artifactId>rpc-interface</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

② 实现服务接口。

```
public class HelloServiceImple implements HelloService {
    public String hello(String name) {
        System.out.println("收到消息: " + name);
        return "你好: " + name;
    }
}
```

③ 编写监听服务类。

```
public class Server {

    private static ExecutorService executor = Executors.newFixedThreadPool(10);

    private static final HashMap<String, Class> serviceRegistry = new HashMap<String, Class>();

    public void register(Class serviceInterface, Class impl) {
        //注册服务
        serviceRegistry.put(serviceInterface.getName(), impl);
    }

    public void start(int port) throws IOException {
        final ServerSocket server = new ServerSocket();
        server.bind(new InetSocketAddress(port));
        System.out.println("服务已启动");
        while (true) {
            executor.execute(new Runnable() {
                public void run() {
                    Socket socket = null;
                    ObjectInputStream input = null;
                    ObjectOutputStream output = null;
                    try {
                        socket = server.accept();
                        // 接收到服务调用请求，将码流反序列化定位具体服务
                        input = new ObjectInputStream(socket.getInputStream());
                        String serviceName = input.readUTF();
                        String methodName = input.readUTF();
                        Class<?>[] parameterTypes = (Class<?>[]) input.readObject();
                        Object[] arguments = (Object[]) input.readObject();
                        // 在服务注册表中根据调用的服务获取到具体的实现类
                        Class serviceClass = serviceRegistry.get(serviceName);
                        if (serviceClass == null) {
                            throw new ClassNotFoundException(serviceName + " 未找到");
                        }
```

```
                    Method method = serviceClass.getMethod(methodName, parameterTypes);
                    // 调用获取结果
                    Object result = method.invoke(serviceClass.newInstance(), arguments);
                    // 将结果序列化后发送回客户端
                    output = new ObjectOutputStream(socket.getOutputStream());
                    output.writeObject(result);
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    // 关闭资源
                    try {
                        if (socket != null) socket.close();
                        if (input == null) input.close();
                        if (output == null) output.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
    }
}
```

register()

提供一个数组保存所注册的服务接口及实现类。

start()

启动一个阻塞式的 Socket 服务用于等待客户端发起的调用请求，当收到请求后将码流反序列化成对象，并根据接口从注册列表中寻找具体实现类，最终通过反射的方式调用该实现类返回结果。

④ 注册服务并启动服务端。

```
public class App {
    public static void main(String[] args) throws IOException {
        Server server = new Server();
        // 注册服务
        server.register(HelloService.class, HelloServiceImple.class);
        // 启动并绑定端口
        server.start(8020);
    }
}
```

1.3.3 客户端

新建用于调用服务的 Maven 应用，并引入刚编写的接口应用依赖。

Maven 参数

```
groupId: org.book
artifactId: rpc-client
version: 0.0.1-SNAPSHOT
packaging: jar
```

① 在 pom.xml 文件中引入依赖。

```
<dependency>
    <groupId>org.book</groupId>
    <artifactId>rpc-interface</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

② 编写远程调用类。

```
public class Client<T> {
    @SuppressWarnings("unchecked")
    public static <T> T get(final Class<?> serviceInterface, final InetSocketAddress addr) {
        T instance = (T) Proxy.newProxyInstance(serviceInterface. getClassLoader(),
new Class<?>[]{serviceInterface},
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                        Socket socket = null;
                        ObjectOutputStream output = null;
                        ObjectInputStream input = null;
                        try {
                            // 连接服务端
                            socket = new Socket();
                            socket.connect(addr);
                            // 将调用的接口类、方法名、参数列表等序列后发送给服务提供者
                            output = new ObjectOutputStream(socket.getOutputStream());
                            output.writeUTF(serviceInterface.getName());
                            output.writeUTF(method.getName());
                            output.writeObject(method.getParameterTypes());
                            output.writeObject(args);
                            // 同步阻塞等待服务器返回应答，获取应答后返回
```

```
                    input = new ObjectInputStream(socket.getInputStream());
                    return input.readObject();
                } finally {
                    if (socket != null) socket.close();
                    if (output != null) output.close();
                    if (input != null) input.close();
                }
            }
        });
        return instance;
    }
}
```

使用 JDK 动态代理方式，根据提供的服务接口类将接口序列化成码流，向目标服务端发起 Socket 远程调用请求，获得服务端反馈的结果并反序列化成对象后返回。

③ 调用测试。

```
public class App {
    public static void main(String[] args) throws IOException {
        HelloService service = Client.get(HelloService.class, new InetSocketAddress
("localhost", 8020));
        System.out.println(service.hello("RPC"));
    }
}
```

运行结果如下所示：

```
// 客户端
hello : RPC

// 服务端
服务已启动
收到消息: RPC
```

本章示例代码详见异步社区网站本书页面。

1.3.4　完善框架

服务之间的调用已基本实现，但想将它投入正式开发使用还有很多细节需要完善。

通信

当请求过大后会发现，BIO（同步阻塞式）的通信方式会消耗过多的资源导致服务器变慢甚至崩溃。

序列化与反序列化

在发起网络请求前，将对象转换成二进制串便于网络传输；收到消息请求后，将二进制串反转换成对象便于后续处理。序列化及反序列化直接影响到整个 RPC 框架的效率及稳定性。

服务注册中心

发起服务调用时，都需要指定服务提供方的访问地址（ip + 端口），如果当前服务提供方有多个或一个服务部署在多个机器上，调用时每次手动指定访问地址非常麻烦，这时就需要一个公共的注册中心去管理这些服务。

负载均衡

实施微服务的目的是为了让系统在进行横向扩展时能够拥有更多的计算资源，如果发现某一提供服务的机器负载较大，这就需要将新的需求转发到其他空闲的机器上。

服务监控

服务提供方有可能崩溃无法继续提供服务，在客户端进行调用时就需要将这些无法使用的服务排除掉。

异常处理

当服务端有异常发生导致无法返回正确的结果时，客户端并不知道该如何处理，只能等待并最终以超时结束此次远程调用请求。

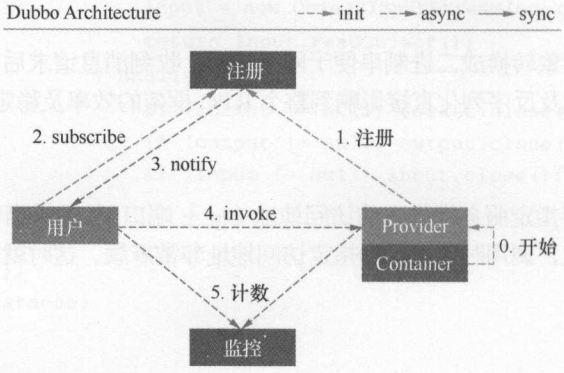
以上所有的问题在后续将要介绍的 Dubbo 与 Spring Cloud 分布式框架中都得到了很好的解决，甚至基于 Spring Boot 构建的应用能让整个开发过程变得轻松愉快。

1.4 主流微服务框架介绍

1.4.1 Dubbo

阿里巴巴在 2011 年开源了 Dubbo 框架，虽然在 2013 年停止更新，但在 2017 年 9 月又重启维护并发布了新版本。目前已有很多的公司将自己的业务建立在 Dubbo 之上，同时阿里云也推出了企业级分布式应用服务 EDAS，为 Dubbo 提供应用托管。

Dubbo 采用 Zookeeper 作为注册中心，RPC 作为服务调用方式，致力于提供高性能和透明化的 RPC 远程服务调用方案。它与 Spring 无缝集成，基于服务提供方（服务端）与服务调用方（客户端）角色构建简单模型，其优点是使用方便、学习成本低。



① 服务提供方发布服务到服务注册中心。

② 服务消费方从服务注册中心订阅服务。

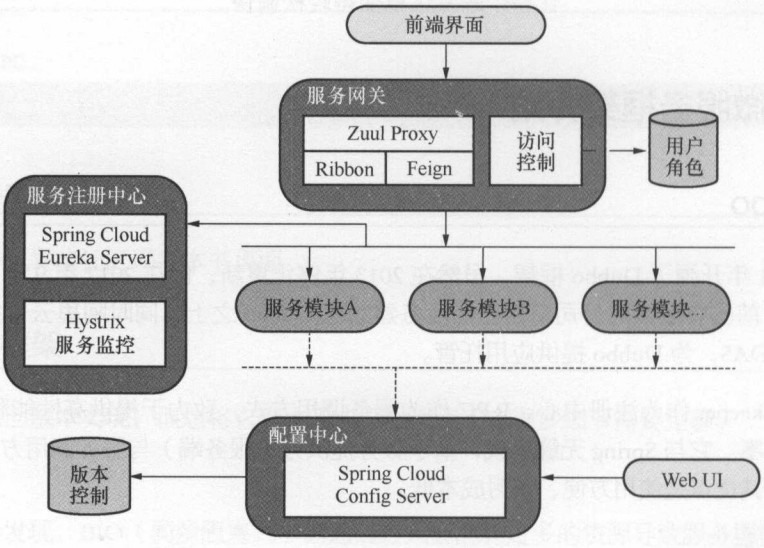
③ 注册中心通知消息调用方服务已注册。

④ 服务消费方调用已经注册的可用服务。

⑤ 监控计数。

1.4.2　Spring Cloud

Spring Cloud 基于 Spring Boot 实现，使用 HTTP 的 RESTful 风格 API 作为调用方式。它所包含的多个子项目共同构建了微服务架构体系。



Netflix Eureka

Spring Cloud 的服务注册中心提供服务注册、服务发现、负载均衡等功能。

Netflix Hystrix

当某个服务发生故障之后，则触发熔断机制（Hystrix）向服务调用方返回结果标识错误，而不是一直等待服务提供方返回结果，这样就不会使得线程因调用故障服务而被长时间占用不释放，避免了故障在分布式系统中的蔓延。

Netflix Zuul

代理各模块提供的服务，统一暴露给第三方应用。提供动态路由、监控、弹性、全等的边缘服务。

Config Server

分布式架构下多微服务会产生非常多的配置文件，分布式配置中心（Config Server）将所有配置文件交由 GIT 或 SVN 进行统一管理，避免出错。

Spring Boot

在使用 Spring 开发时，通常需要完成 Spring 框架及其他第三方工具配置文件的编写，非常麻烦。Spring Boot 通过牺牲项目的自由度来减少配置的复杂度，约定一套规则，把这些框架都自动配置集成好，从而达到“开箱即用”。

第2章　模块拆分

2.1　拆分逻辑

2.2　单模块

2.3　基础模块

2.4　复杂模块

将完整地使用本地调用方式的垂直应用拆分成多个微小的服务，每个服务模块负责提供各自独立的服务接口，并通过网络调用的方式将各个服务模块组织起来形成完整的微服务系统。

这里介绍微服务架构中拆分模块的基本逻辑，更为完善的模块拆分可以基于领域驱动设计（Domain-Driven Design，DDD）进行。

2.1 拆分逻辑

模块拆分是分布式微服务实施时的困难之一，它将直接影响到系统的复杂度、团队协作、代码维护难度、硬件资源分配等方面。模块拆分得越细，则能够更灵活地分配硬件资源与更方便地进行团队协作，但这样也会增加系统复杂度与代码维护难度，在团队人数较少的情况下无疑增加了负担。拆分模块时需要以具体的业务需求与系统请求压力分布为出发点进行权衡取舍。

系统复杂度

业务的复杂性决定了被拆分的模块之间必然存在一定的依赖，模块被拆分得越细就意味着会产生更多的依赖关系，在拆分解耦的同时必然增加了整个系统的复杂度。

随着业务的丰富不可避免地使系统越来越复杂，我们没有办法拒绝复杂但应通过规范、约定、框架等手段尽量做到结构及代码的清晰与整洁。

团队协作

当垂直应用变得庞大且复杂需要更多的工程师维护时，一般会使用 Maven 的多模块依赖特性将单一的应用拆分成多个模块，每个模块分给不同的工程师维护，最终团队成员提交模块，根据依赖关系打包成一个应用发布。

微服务天生由多模块组成，各个模块交由具体的专人负责，团队之间通过模块所暴露的服务进行协作。拆分模块的同时也确定了团队协作的方式。

代码维护难度

微服务能够解决在垂直应用中错综复杂的业务逻辑耦合在一起的维护困难问题，但也并不是将模块拆分得越细越好，过多的模块反而会增加工作量与代码维护难度。

每个模块都处理着属于自己的业务逻辑，它们提供服务并维护着与其他模块的依赖关系，如果服务提供方的返回结果发生了变动，则各个调用方均要修改自己的代码。

在实际开发中不可避免地要跨模块调试，调试过程中所涉及的模块数量越多，则整个过程就越麻烦。

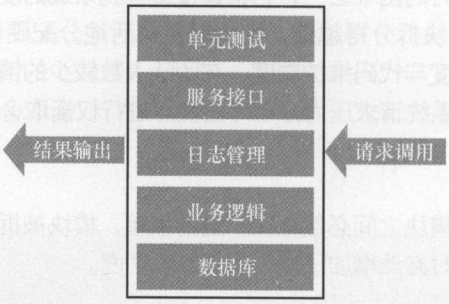
硬件资源分配

系统中存在请求压力不均衡的情况，模块拆分得越细，则能更有针对性地为高压力模块分配更多的计算资源，避免浪费。

2.2　单模块

为了设计出低耦合、高内聚的系统，需要确保每个模块都具有一定的独立性，每个模块只完成它所负责的业务功能，并且模块之间做到最少联系及接口简单。模块之间的边界则是思考的重点。

单个模块内聚了相关性较强的功能，并且拥有独立的数据库（ORM）、单元测试、运行内存、业务逻辑处理等。可以将单个模块看成是一个完整的垂直应用，只是在输入（接受请求）、输出（结果输出）时有差别。

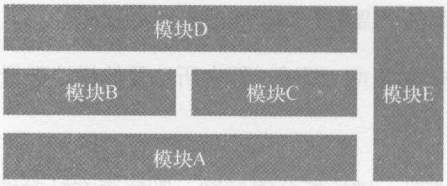


大部分请求都是同步的，即消费方发起请求后等待提供方完成计算并返回结果，但如果等待的过程过于漫长，则需要借助消息队列与回调将请求的过程变为异步，消费方向服务提供方的消息队列中发送请求消息，服务方计算完成后调用消费方的方法通知结果。

> 异步请求适合处理批量处理类的功能，例如群发邮件，一次性发送 1000 封邮件需要一定的发送时间，采用异步处理，则消费方只需向发送邮件的服务队列中增加收件列表，任务完成后调用消费方的接口告知发送结果即可。

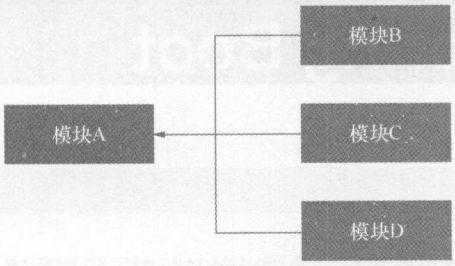
2.3　基础模块

一个复杂的业务功能需要依赖多个模块所提供的功能实现，在进行模块拆分时需要提前对业务所涉及的模块进行梳理，最终如图所示，各个模块由上至下依次层级依赖。



各个模块提供的服务具有一定的原子性，保持独立不重叠。将不需要依赖其他模块或依赖较少的模块抽象为基础模块，为更为复杂的业务逻辑做准备，提高复用性与扩展性。

2.4 复杂模块



为实现复杂业务将基础模块进行聚合重组时，每个模块对自身数据库操作事务的管理比较简单，但基于网络跨模块的二阶事务管理则会将整个过程变得无比复杂，并且耗费更多的资源，所以在进行模块拆分时且尽量规避二阶事务的产生。在无法避免分布式事务的情况下，可以采用 TCC（Trying Confirming Canceling）补偿性事务解决方案实现对二阶事务的管理。

垂直应用在处理购物车或登录状态管理等功能时一般会基于 Session 实现，但在分布式架构下 Session 的共享与传递会增加整个系统的耦合度并且提高复杂性。模块在处理自身业务逻辑及服务调用时，尽量以无状态协议的角度进行设计，请求完立刻释放资源。维护状态的工作可由服务提供方增加状态检查的服务，但面对高查询、低修改的场景时可以基于约定交由公共缓存系统（Redis）维护。

复杂的订单模块为了产生一张订单，需要用户模块提供的买卖双方信息，产品模块提供的货物信息，财务模块提供的账户余额、消息模块提供的短信通知等基础服务。

第3章 Spring Boot

3.1 目录结构

3.2 主要文件

3.3 编辑器集成

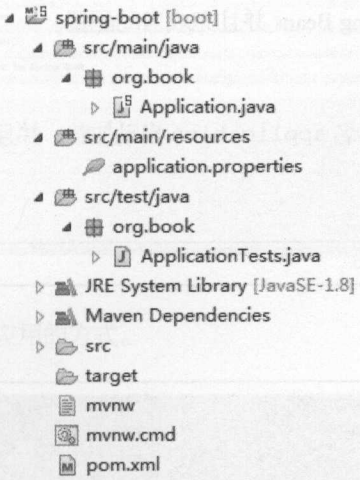
Spring Boot 是 Spring 官方的顶级项目之一，基于 Spring Platform 对 Spring 框架和第三方库进行处理，提供默认配置以降低使用复杂度，可轻松创建单独运行的、基于生产级的 Spring 应用程序。

后续将要介绍的 Spring Cloud 分布式微服务框架也是在 Spring Boot 的基础上构建的，并且 Dubbo 框架的社区也提供了 Spring Boot 的支持。为了更加方便愉快地开发，后续所有例子都将基于 Spring Boot 进行讲解。

3.1 目录结构

Spring Boot 基于 Maven 构建，官网提供了快速初始化服务，只需提供相关 Maven 信息及需要引入的第三方依赖包，便可自动生成应用并打包成 zip 压缩包下载。

将下载好的 zip 压缩包解压后得到的目录如下：



src/main/java

用于存放源代码文件。

src/main/resources

用于存放配置文件，如果在使用 Spring Initializr 创建应用时勾选了 `spring-boot-starter-web` 依赖，则会在此目录自动创建 `static` 目录用于存放静态文件及 `templates` 目录用于存放界面模板文件。

src/test/java

用于存放测试文件。

target

用于存放 Maven 编译后的文件。

3.2　主要文件

Application.java

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

通过注解 `@SpringBootApplication` 指定该类为 Spring Boot 启动类，Spring Boot 将根据该类所在包路径自动扫描子路径下所有的 Spring Beans 并且完成自动配置。

application.properties

Spring Boot 的默认配置文件，根据文件名 `application` 自动加载，并且支持以 `.yml` 为后缀的配置文件。

application.properties 示例

```
spring.application.name=demo
```

application.yml 示例

```
spring:
  application:
    name: demo
```

pom.xml

Maven 用于管理依赖的重要文件，Spring Boot 基于 Maven 的项目集成特性在 `parent` 节点中配置了相关信息，同时指定了 `spring-boot-maven-plugin` 插件用于编译 Spring Boot 应用。

mvnw

对 Maven 的 mvn 命令的封装，`mvnw.cmd` 用于 Windows 系统，而 `mvnw` 则用于 Linux 系统。

3.3　编辑器集成

目前主流的编辑器如 Eclipse、IntelliJ IDEA 等都集成了 Spring Initializr 用于创建 Spring Boot 应用，

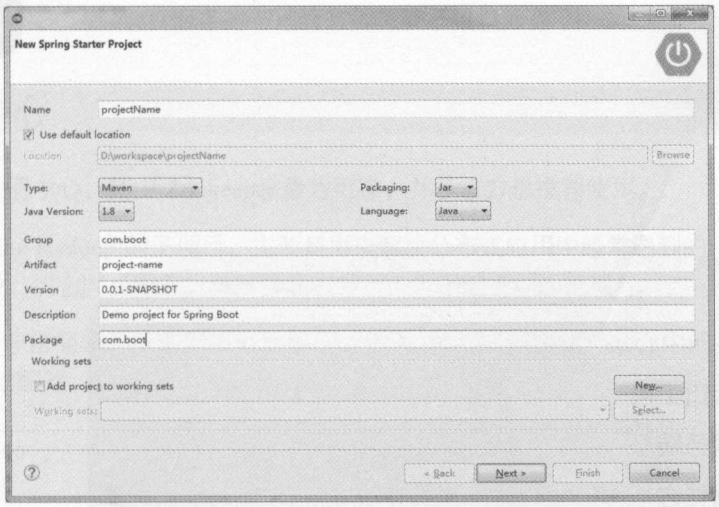
并且默认支持 Maven，使整个开发过程变得轻松愉快。

Eclipse

使用插件的方式集成 Spring Initializr

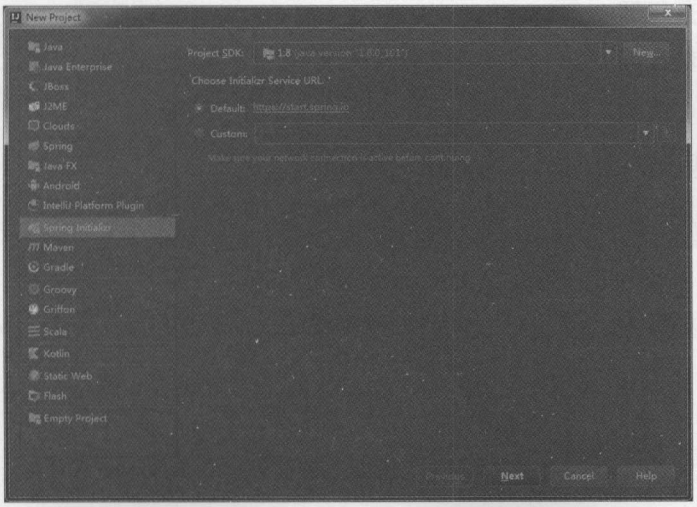
插件名称：Spring Tools（aka Spring IDE and Spring Tool Suite）

安装方式：菜单栏 → Helps → Eclipse Marketplace → 搜索 STS



IntelliJ IDEA

已默认集成 Spring Initializr 新建应用的方式。



第4章 Dubbo

4.1 注册中心
4.2 接口工程
4.3 服务端
4.4 消费方
4.5 网关
4.6 监控中心
4.7 服务管理
4.8 负载均衡
4.9 服务降级
4.10 集群容错

最基本的 Dubbo 工程由服务提供方、消费方、服务接口组成，接口工程中编写所提供服务的接口（Interface）由服务提供方实现具体业务逻辑并注册服务，消费方则基于接口工程中所规定的服务接口进行调用，各工程之间基于 Maven 管理依赖。

服务方与消费方都依赖于接口服务

4.1 注册中心

Dubbo 支持多种注册中心，其中 Zookeeper 最为可靠，并且官方也推荐使用。

Zookeeper 是 Apache Hadoop 的子项目，主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

Zookeeper 的安装非常简单，从官网下载后解压，进入 conf 目录中把 zoo_sample.cfg 重命名为 zoo.cfg 便可开始配置。

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=E:\\apache\\zookeeper-3.5.3-beta\\data
clientPort=2181
admin.serverPort=9090
```

tickTime

服务器之间或客户端与服务器之间维持心跳的时间间隔。

initLimit

配置在集群中与其他 Zookeeper 的连接最大心跳时间间隔数。

syncLimit

标识 Leader 与 Follower 之间发送消息时请求和应答的时间长度，规定了在此期间最长不能超过多少个心跳数。

dataDir

保存数据的目录。

clientPort

客户端连接服务器的端口，zookeeper 会监听这个端口，接受客户端的访问请求。

admin.serverPort

Jetty 服务的监听端口，默认是 8080。

集群配置

server.A：B:C:D，其中 A 表示服务器编号，B 表示该服务器的 IP 地址，C 和 D 是两个 TCP 端口号，分别用于仲裁和 Leader 选举。

server.1:192.169.1.22:2222:2223

server.2:192.169.1.23:2222:2223

示例表示当前的 Zookeeper 与 IP 为 22 和 23 的 Zookeeper 组成集群，如果 IP 相同，则用于仲裁和选举的端口号需要区分。

启动

启动脚本存放在 zookeeper 的 bin 目录中，根据你的操作系统选择。

Windows：由 CMD 或 PowerShell 命令进入 Zookeeper 的 bin 目录中，并执行 `.\zkServer.cmd`。

Linux：进入 Zookeeper 的 bin 目录中，并执行 `zkServer.sh start`。

4.2 接口工程

与之前“实现一个最简单的微服务框架”中的公共接口一样，这里需要新建一个 Maven 应用，并服务于服务提供方与消费方。

Maven参数

```
groupId: org.book.rpc.dubbo
artifactId: dubbo-api
version: 0.0.1-SNAPSHOT
packaging: jar
```

编写接口

```
public interface IHello {
    public String say(String msg);
}
```

4.3 服务端

① 新建 Spring Boot 应用。

在 Eclipse 中可以使用 STS 插件新建，地址为：菜单栏 → File → New → Spring Starter Project

Maven参数

```
groupId: org.book.rpc.dubbo
artifactId: dubbo-service
version: 0.0.1-SNAPSHOT
packaging: jar
```

② 在 pom.xml 文件中添加 Dubbo 与接口应用的依赖。

```
<dependency>
  <groupId>io.dubbo.springboot</groupId>
  <artifactId>spring-boot-starter-dubbo</artifactId>
  <version>1.0.0</version>
</dependency>
<dependency>
  <groupId>org.book.rpc.dubbo</groupId>
  <artifactId>dubbo-api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

③ 编写代码实现接口。

```
@Service
public class HelloImple implements IHello {
    @Override
    public String say(String msg) {
        System.out.println("你好: " + msg);
        return msg;
    }
}
```

Dubbo 的 @Service 注解用于暴露服务，其可配置的属性如下。

version

指定该服务的版本。

group

服务所属分组，当一个接口有多种实现时，可以用 group 区分。

cache

该服务的缓存策略。

- lru 基于“最近最少使用”原则删除多余缓存，保持最热的数据被缓存。
- threadlocal 当前线程缓存，比如一个页面渲染，用到很多 portal，每个 portal 都要去查用户信息，通过线程缓存，可以减少这种多余访问。

async

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务的功能，相对多线程开销较小。

delay

延迟暴露服务，如果你的服务需要 warmup 时间，比如初始化缓存、等待相关资源就位等，可以使用 delay 进行延迟暴露。为 -1 时，则表示延迟到 Spring 初始化完成后，再暴露服务 (基于 Spring 的 ContextRefreshedEvent 事件触发暴露)。

timeout

调用服务时的超时时间，单位为秒。

mock

设为 true，表示使用缺省 Mock 类名，即接口名 + Mock 后缀，服务接口调用 Mock 实现类，该 Mock 类必须有一个无参构造函数，与 Local 的区别在于，Local 总是被执行，而 Mock 只在出现非业务异常 (比如超时、网络异常等) 时执行；Local 在远程调用之前执行，Mock 在远程调用后执行。

retries

远程服务调用重试次数，不包括第一次调用，不需要重试设为 0。

token

令牌验证。如果为空，表示不开启；如果为 true，表示随机生成动态令牌；否则使用静态令牌。令牌的作用是防止消费者绕过注册中心直接访问，保证注册中心的授权功能有效，如果使用点对点调用，需关闭令牌功能。

dynamic

服务是否动态注册，如果设为 false，注册后将显示 disable 状态，需人工启用，并且服务提供者停止时，也不会自动取消册，需人工禁用。

register

该协议的服务是否注册到注册中心。

deprecated

服务是否过时，如果设为 true，消费方引用时将打印服务过时警告 error 日志。

accesslog

设为 true，将向 logger 中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件。

executes

在暴露服务中的每个方法服务器端并发执行（或占用线程池中的线程数）。

actives

在暴露服务中的每个方法客户端并发执行（或占用连接的请求数）。

loadbalance

负载均衡策略，可选值为 random、roundrobin、leastactive，分别表示随机、轮循、最少活跃调用。

此参数均为小写。

connections

限制客户端服务使用连接数（如果是长连接，比如 Dubbo 协议，connections 表示该服务对每个提供者建立的长连接数）。

protocol

该服务所使用的协议，不同服务在性能上适用不同协议进行传输，比如大数据用短连接协议，小数据大并发用长连接协议，默认为 Dubbo。

- Dubbo 协议，采用单一长连接和 NIO 异步通信，适合于“小数据量大并发”的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。
- Hessian 协议，传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。
- RMI 协议，采用 JDK 标准的 java.rmi 实现，常规远程服务方法调用，与原生 RMI 服务互操作。
- HTTP 协议，可用浏览器查看，通过表单或 URL 传入参数。
- WebService 协议，SOAP 文本序列化，多用于系统集成，跨语言调用。
- Thrif 协议，对 thrift（Facebook 的 RPC 框架）的原生协议的扩展。

④ 在 appaction.properties 文件中配置 Dubbo。

```
spring.dubbo.application.name=provider
spring.dubbo.registry.address=zookeeper://127.0.0.1:2181
spring.dubbo.protocol.name=dubbo
spring.dubbo.protocol.port=-1
spring.dubbo.scan=org.book
```

spring.dubbo.application.name

当前应用名称，用于注册中心计算应用间依赖关系。

spring.dubbo.application.version

当前应用的版本。

spring.dubbo.application.logger

日志输出方式，可选：slf4j、jcl、log4j、jdk。

spring.dubbo.registry.address

服务所使用的注册中心，A://B:C，其中 A 为注册中心类型，B 为注册中心所在 ip 地址，C 为注册中心的端口号。配置集群时候以逗号分隔。例如 A://B1:C1,A://B2:C2

- Zookeeper 注册中心：使用 zookeeper 作为注册中心。
- Multicast 注册中心：以广播的方式在地址段 224.0.0.0 – 239.255.255.255 实现服务的注册与发现，适合小规模应用或开发阶段使用。
- Redis 注册中心：使用 redis 作为注册中心，通过心跳的方式检测脏数据，服务器时间必须相同，并且对服务器有一定压力。
- Simple 注册中心：注册中心本身就是一个普通的 Dubbo 服务，可以减少第三方依赖，使整体通信方式一致。

spring.dubbo.registry.username

登录注册中心用户名，如果注册中心不需要验证可不填。

spring.dubbo.registry.password

登录注册中心密码，如果注册中心不需要验证可不填。

spring.dubbo.registry.timeout

注册中心请求超时时间 (毫秒)。

spring.dubbo.registry.register

是否向注册中心注册服务，如果设为 false，将只订阅，不注册。

spring.dubbo.registry.subscribe

是否向注册中心订阅服务，如果设为 false，将只注册，不订阅。

spring.dubbo.registry.transporter

网络传输方式，可选 mina、netty。

spring.dubbo.protocol.name

服务所使用的协议名称。

spring.dubbo.protocol.port

服务提供方所暴露的端口号，多个服务提供方不可重复。

Dubbo 协议缺省端口为 20880，RMI 协议缺省端口为 1099，HTTP 和 Hessian 协议缺省端口为 80。如果配置为 -1 或者没有配置，则会分配一个没有被占用的端口。

spring.dubbo.scan

扫描服务所在包路径。

spring.dubbo.protocol.threadpool

线程池类型，可选：fixed、cached

spring.dubbo.protocol.threads

服务线程池大小 (固定大小)。

spring.dubbo.protocol.iothreads

IO 线程池大小 (固定大小)。

spring.dubbo.protocol.accepts

服务提供方最大可接受连接数。

spring.dubbo.protocol.payload

请求及响应数据包大小限制，单位为字节，默认为 8838860B (8MB)。

spring.dubbo.protocol.serialization

协议序列化方式，当协议支持多种序列化方式时使用，比如：Dubbo、Hessian2、Java 原生，以及 HTTP 协议的 JSON 等。

4.4 消费方

① 新建 Spring Boot 应用。

Maven参数

```
groupId: org.book.rpc.dubbo
artifactId: dubbo-client
version: 0.0.1-SNAPSHOT
packaging: jar
```

② 在 pom.xml 文件中添加 Dubbo 与接口工程的依赖。

```
<dependency>
  <groupId>io.dubbo.springboot</groupId>
  <artifactId>spring-boot-starter-dubbo</artifactId>
  <version>1.0.0</version>
</dependency>
<dependency>
  <groupId>org.book.rpc.dubbo</groupId>
  <artifactId>dubbo-api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

③ 编写远程调用 Dubbo 服务。

```
@Component
public class InvokeService {
    @Reference
    public IHello hello;
}
```

Dubbo 的 @Reference 注解可以用于生成远程服务代理，其可配置的属性如下。

version

服务版本，与服务提供者的版本一致。

group

服务分组，当一个接口有多个实现时，可以用分组区分，必须和服务提供方一致。

timeout

服务方法调用超时时间 (毫秒)。

retries

远程服务调用重试次数，不包括第一次调用，不需要重试请设为 0。

connections

设置提供者的最大连接数。

loadbalance

负载均衡策略，可选值为 random、roundrobin、leastactive，分别表示：随机、轮循、最少活跃调用。

此参数均为小写。

async

是否异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程。

check

启动时检查提供者是否存在，true 报错，false 忽略。

init

是否等到有服务注入或引用该实例时再初始化。

protocol

只调用指定协议的服务提供方，其他协议忽略。

cache

以调用参数为 key，缓存返回结果，可选：lru、threadlocal、jcache 等。

④ 在 appaction.properties 文件中配置 Dubbo。

```
spring.dubbo.application.name=consumer
spring.dubbo.registry.address=zookeeper://127.0.0.1:2181
spring.dubbo.scan=org.book
```

与服务提供方的配置一致，指定与服务提供方一致的注册中心地址，并提供不同的 application.name 便可完成调用端的配置。

⑤ 测试调用。

在 Spring Boot 的入口处获得刚编写的用于调用 Dubbo 服务的 InvokeService，并依次启动 Zookeeper、服务方、消费方工程完成远程调用。

```
@SpringBootApplication
public class DubboClientApplication {
```

```
        public static void main(String[] args) {
            ConfigurableApplicationContext run = SpringApplication.run(DubboClient Application.class,
args);
            InvokeService service = run.getBean(InvokeService.class);
            System.out.println("收到返回结果: "+service.hello.say("rpc"));
        }
    }
```

4.5 网关

模块之间互相调用时，为了降低由网络波动带来的不确定性因素并提升系统安全性，生产环境中所有模块一般都运行在内网环境中，并单独提供一个工程作为网关服务，开放固定端口代理所有模块提供的服务，并通过拦截器验证所有外部请求以达到权限管理的目的。外部应用有可能是 App、网站或桌面客户端，为了达到通用性，网关服务一般为 Web 服务，通过 HTTP 协议提供 RESTful 风格的 API 接口。

① 新建 Spring Boot 应用。

Maven参数

```
    groupId: org.book.rpc.dubbo
    artifactId: dubbo-getway
    version: 0.0.1-SNAPSHOT
    packaging: jar
```

② 在 pom.xml 文件中添加 Dubbo 与接口工程的依赖。

```
    <dependency>
      <groupId>io.dubbo.springboot</groupId>
      <artifactId>spring-boot-starter-dubbo</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.book.rpc.dubbo</groupId>
      <artifactId>dubbo-api</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

spring-boot-starter-web

支持 Web 应用开发，包含 Tomcat 和 Spring-MVC。

③ 在 appaction.properties 文件中配置。

```
spring.dubbo.application.name=geteway
spring.dubbo.registry.address=zookeeper://127.0.0.1:2181
spring.dubbo.registry.register=false
spring.dubbo.scan=org.book
server.port=8081
```

网关服务只负责调用模块的服务，通过 spring.dubbo.registry.register=false 限制只调用服务，并不注册。

server.port

网关服务所访问端口号

④ 编写用于调用 Dubbo 服务的编写控制器。

```
@RestController
public class RpcController {

    @Reference
    private IHello hello;

    @RequestMapping(value = "/")
    public String say() {
        return hello.say("rpc");
    }
}
```

使用注解 @RestController 标识当前类为一个 Servlet，并通过注解 @RequestMapping 映射请求，相当于 Servlet 在 web.xml 的配置。

@RestController 继承自 @Controller，它将自动把 response 的返回结果进行 json 序列化，并且可以为请求链接增加 .json 或 .xml 后缀来指定序列化方式。如果想使用 @Controller 达到返回 json 结构的结果，则需要为每个 @RequestMapping 增加 @ResponseBody 注解。

⑤ 编写限验证逻辑。

Dubbo 中所暴露的接口由网关服务转为 HTTP 协议的 API 调用，这些接口通常需要安全检查，所以这里新建 HandlerInterceptor 拦截器的实现类用来模拟最简单的权限验证。

```
public class RequestInterceptor implements HandlerInterceptor {

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object obj,
```

```
            Exception exception) throws Exception {
        }

        @Override
        public void postHandle(HttpServletRequest request, HttpServletResponse response, Object obj,
                ModelAndView modelAndView) throws Exception {
        }

        @Override
        public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object obj) throws Exception {
            String token = request.getParameter("token");
            if (token != null && token.equals("1"))
                return true;
            response.getWriter().write("token error");
            return false;
        }
    }
```

实现 HandlerInterceptor 接口后，Spring 提供了 3 个方法用于在不同阶段满足过滤的需求。

afterCompletion()

当前对应的 Interceptor 的 preHandle 方法的返回值为 true 时执行，主要用于资源清理工作。

postHandle()

当前请求进行处理之后执行，主要用于日志记录、权限检查、性能监控、通用行为等。

preHandle()

在请求处理之前执行，主要用于权限验证、参数过滤等。

Spring 允许多个拦截器同时存在，通过拦截器链管理。

返回值为 true 时执行下一个拦截器，直到所有拦截器执行完，再运行被拦截的 Controller。

返回值为 false 时不再执行后续的拦截器链及被拦截的 Controller。

⑥ 配置拦截器。

```
    @Configuration
    public class WebAppConfig extends WebMvcConfigurerAdapter {
        @Override
        public void addInterceptors(InterceptorRegistry registry) {
            registry.addInterceptor(new RequestInterceptor()).addPathPatterns("/**");
            super.addInterceptors(registry);
```

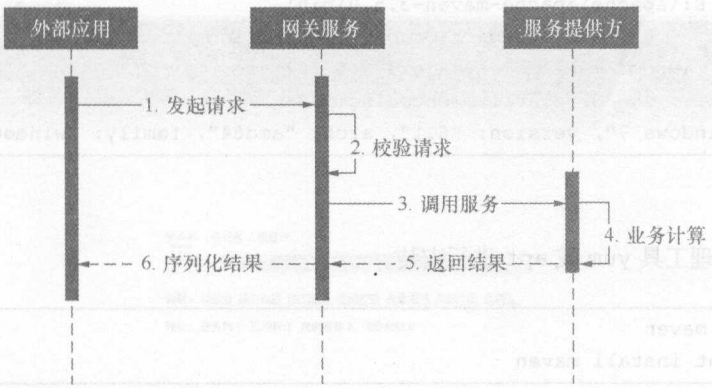
```
    }
}
```

WebMvcConfigurerAdapter用于进行WebMVC环境中的相关配置，通过对addInterceptors()方法重写可以使之前编写好的拦截器生效。@Configuration注解则用于标识当前为一个配置类，在Spring Boot启动时会被加载。

使用addPathPatterns方法设置拦截规则，而excludePathPatterns方法则可以设置排除的拦截规则。

此处设置的addPathPatterns("/**")表示拦截所有请求，如果只拦截account目录下的请求，则写为/account/**。

⑦ 测试调用。



依次启动Zookeeper、服务提供方、网关应用。

访问http://localhost:8081：由于缺少token参数，拦截器生效，请求被拦截则直接返回token error信息。

访问http://localhost:8081?token=1：调用服务并返回结果。

4.6 监控中心

Dubbo官方提供了dubbo-admin子项目，主要用于路由规则、动态配置、服务降级、访问控制、权重调整、负载均衡等管理。

① 从官网下载Dubbo源码。

```
git clone https://github.com/alibaba/dubbo.git
```

② 编译 Dubbo。

Dubbo 也是基于 Maven 构建，所以这里需要先在开发环境中安装 Maven。

Windows安装

将下载好的 maven 包解压后获得 bin 目录地址，添加进 Windows 环境变量中

右键我的电脑 → 属性 → 高级系统设置 → 高级 → 环境变量 → 系统变量 → 双击变量名为 path 的变量。

在末尾增加“;maven 的 bin 目录路径”，其中；为分隔符，用于区分多个环境变量。

添加完成后在命令行工具中使用 mvn -v 命令查看 Maven 版本。

```
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-04T03:39:06+ 08:00)
Maven home: E:\apache\apache-maven-3.5.0\bin\..
Java version: 1.8.0_101, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_101\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

Linux安装

可直接使用软件管理工具 yum 或 apt 进行安装。

```
yum install maven
    apt-get install maven
```

安装完成后通过 mvn -v 验证安装结果。

当 Maven 安装配置完成后便可进入已下载的 Dubbo 根目录编译源码，通过命令 DskipTests 跳过测试阶段。

```
mvn package -DskipTests
```

③ 配置及启动监控中心。

当项目编译成功后，进入 dubbo-admin 子项目的 target 目录，将编译好的 dubbo-admin-2.5.5-SNAPSHOT.war 解压后移入 Tomcat 的 webapps/ROOT 目录，并修改 dubbo-admin 的 WEB-INF 目录下 dubbo.properties 文件来进行配置。

```
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.admin.root.password=root
dubbo.admin.guest.password=guest
```

dubbo.registry.address

注册中心地址。

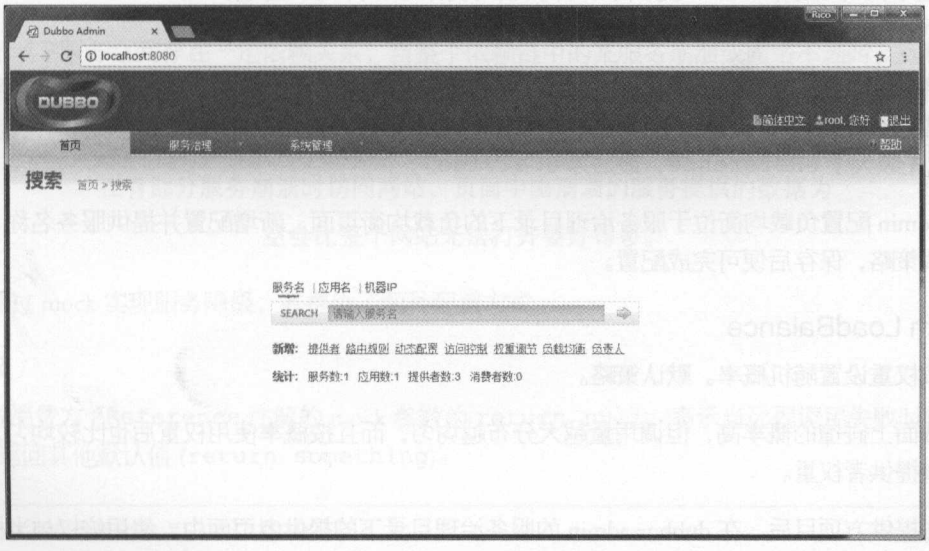
dubbo.admin.root.password

root 管理员的登录密码。

dubbo.admin.guest.password

游客的登录密码。

依次启动 Zookeeper、Tomcat 后，便可在浏览器中访问监控界面。



4.7 服务管理

启动服务提供方或服务消费方项目，便可在服务治理目录下的提供者与消费者页面中查看对应的项。

服务

服务提供方中所暴露的具体服务，与注解 `@Service` 相关。

应用

服务提供方或消费方的名称，由 `appaction.properties` 配置文件中的 `spring.dubbo.application.name` 参数设置。

机器

服务提供方或消费方所运行的环境，由 ip 地址与端口号进行区分。

一个机器可以部署多个应用，而一个应用可以提供多个服务。

4.8 负载均衡

当并发请求与计算的需求越来越大，一台机器难以处理时，将提供该服务的应用部署在多个机器上，共同向注册中心提交服务，用户发起请求调用服务时，则从注册中心根据负载策略寻找服务提供方，最终完成此次调用。

为了适应不同的场景，Dubbo 提供了 4 种负载均衡策略，并可以通过实现 `com.alibaba.dubbo.rpc.cluster.LoadBalance` 接口扩展自定义策略。

配置策略时可以在服务提供方（`@Service` 注解）与消费方（`@Reference` 注解）修改 `LoadBalance` 参数，也可以在 dubbo-admin 中使用图形化界面完成配置。当同时配置了不同的策略时，dubbo-admin 优先级大于 `@Reference`，最后为 `@Service`。

dubbo-admin 配置负载均衡位于服务治理目录下的负载均衡页面。新增配置并提供服务名称、方法名称、负载策略，保存后便可完成配置。

Random LoadBalance

随机，按权重设置随机概率。默认策略。

在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

启动服务提供方项目后，在 dubbo-admin 的服务治理目录下的提供者页面中，使用倍权与半权功能调节权重值。

> 例如将同一服务部署两次，会得到两个服务提供者，分别设置权重值为 100 与 200，假设当前有 3000 个并发请求，权重值为 100 的服务将接受 1000 个请求，而权重值为 200 的则接受 2000 个请求。

RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。

存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没崩溃。当请求调到第二台时就卡在那里，久而久之，所有请求都卡在第二台机器上。

LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。

使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发给同一提供者。

当某一台提供者崩溃时，原本发往该提供者的请求，基于虚拟节点被平摊给其他提供者，不会引起剧烈变动。

4.9 服务降级

分布式环境下服务之间存在一定依赖关系，当整个依赖链中的某服务崩溃或网络不通时，远程服务无法调用成功并抛出 RpcException 异常。为了避免由一个服务崩溃而引起的连锁反应以及保持主业务的正常运行，可以对服务进行降级处理，在调用失败后返回默认数据。

在有部分服务崩溃时访问网站，页面中由崩溃的服务提供的数据为空会比整个网站无法打开要好得多。

Dubbo 通过 mock 实现服务降级，并提供了两种配置方式。

简单降级

设置服务消费方 @Reference 注解的 mock 参数为 return null，表示当远程调用失败时直接返回 null，或返回其他默认值 (return something)。

```
@Reference(mock = "return null")
private IHello hello;
```

复杂降级

mock 可以指定一个具体的类，当调用失败时执行，以满足复杂的业务逻辑。mock 类需要实现服务接口，类名规范为：接口名 +Mock。

① 在接口工程中创建 mock 类。

```
public class IHelloMock implements IHello {
    public String say(String msg) {
        return "降级数据";
    }
}
```

② 在服务提供方，配置注解 @Service 指定 mock 类所在位置。

```
@Service(mock = "org.book.service.IHelloMock")
public class HelloImple implements IHello {

    @Override
    public String say(String msg) {
        try {
            // 模拟请求超时
            Thread.sleep(10 * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return msg;
    }
}
```

③ 在服务调用方，配置注解 @Reference 设置 mock 为 true 开启降级。

```
@Reference(mock = "true")
private IHello hello;
```

4.10 集群容错

为了提高系统可用性，将服务提供方部署成多个组成服务集群，在消费方调用服务失败或超时，Dubbo 可以配置多种策略重新调用集群中的其他服务提供方。

在提供方的 @Service 和消费方的 @Reference 注解中配置 cluster 参数，或者用 application.properties 中的 spring.dubbo.service.cluster 参数来指定容错策略。

Failover Cluster

失败自动切换，当出现失败时，重试其他服务器。(默认)

通常用于读操作，但重试会带来更长延迟。

可以通过在服务提供方的 @Service 或消费方的 @Reference 注解中设置 retries 参数来指定重试次数（不含第一次）。

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。

通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。

通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。

通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。

通常用于实时性要求较高的读操作，但需要浪费更多服务资源。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。

通常用于通知所有提供者更新缓存或日志等本地资源信息。

本章示例代码详见异步社区网站本书页面。

第5章 Spring Cloud

5.1 注册中心
5.2 注册服务
5.3 调用服务
5.4 Zuul 网关
5.5 Hystrix 断路器
5.6 服务监控
5.7 应用监控
5.8 熔断器监控
5.9 统一管理配置文件

Spring Cloud 是基于 Spring Boot 的一整套实现微服务的框架。它提供了微服务开发所需的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等组件。

Spring Cloud 有众多的子项目，各自之间都有自己的版本号。为了方便统一由 spring-cloud-dependencies 进行管理，使用时只需引入具体的依赖而不需要提供版本号。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

dependencyManagement 是 Maven 用于管理依赖的一种方式，
它只申明依赖但并不直接引入。

Spring Boot 与 Spring Cloud 版本依赖关系如下。

- Angel 版本对应 Spring Boot 1.2.x
- Brixton 版本对应 Spring Boot 1.3.x
- Camden 版本对应 Spring Boot 1.4.x
- Dalston 版本对应 Spring Boot 1.5.x

后续示例均基于 Dalston.RELEASE 版本进行。

5.1 注册中心

与 Dubbo 一样，Spring Cloud 支持 Zookeeper 作为注册中心，但官方更推荐 Spring Cloud Netflix 的 Eureka。

Spring Cloud Netflix 是 Spring Cloud 的子项目之一，主要内容是对 Netflix 公司一系列开源产品的包装，它为 Spring Boot 应用提供了自配置的 Netflix OSS 整合。通过一些简单的注解，开发者就可以快速地在应用中配置一些常用模块并构建庞大的分布式系统。它主要提供的模块包括：服务发现（Eureka）、

断路器（Hystrix）、智能路由（Zuul）、客户端负载均衡（Ribbon）等。

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Server 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-eureka
version: 0.0.1-SNAPSHOT
packaging: jar
```

pom依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

② Spring Boot 入口开启 Eureka 服务。

```
@SpringBootApplication
@EnableEurekaServer
public class CloudEurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(CloudEurekaApplication.class, args);
    }
}
```

@EnableEurekaServer

Spring Boot 封装了 Eureka，通过该注解将本应用变为 Eureka 服务器。

③ 在 application.properties 配置 Eureka。

```
server.port=8082
eureka.instance.prefer-ip-address=true
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.service-url.defaultZone=http://localhost:${server.port}/eureka/
```

server.port

该服务访问端口。

eureka.instance.hostname

使用主机名广播服务。

eureka.instance.prefer-ip-address

使用 ip 地址广播服务。

eureka.client.register-with-eureka

是否注册到 eureka 服务器。由于应用本身就是注册中心，所以设置为 false。

eureka.client.fetch-registry

是否从 eureka 服务器检索服务。由于本应用的职责就是维护服务实例，所以设置为 false。

eureka.client.service-url

用于服务注册和服务检索的地址。其中 `service-url` 后面需要的是一个 `Map<String,String>` 类型参数，所以 `defaultZone` 可以自己定义。

`${server.port}` 使用 Spring EL 来获取上面已配置的 `server.port` 参数值。

可以通过设置 `http://username:password@localhost:${server.port}/eureka/` 为注册与获取服务增加 HTTP 基础身份验证。

eureka.server.enable-self-preservation

是否关闭 eureka 自我保护模式。当网络产生波动无法请求成功但服务提供方并未崩溃时，自我保护模式并不会删除已在 Eureka 注册表中注册的服务。在对应用频繁的开发环境中建议关闭，生产环境中开启。

eureka.server.eviction-interval-timer-in-ms

服务清理间隔（单位：毫秒，默认：60*1000）。

eureka.instance.lease-renewal-interval-in-seconds

设置服务租约时间（单位：秒，默认：30）。开发环境中可以降低此参数以加快服务注册过程，但生产环境中建议保持默认值。

集群配置

多个 Eureka Server 之间通过 `eureka.client.service-url` 互相注册便可实现集群部署。

Eureka Server 1 配置

```
server.port=8081
eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=http://localhost:8081/eureka/
```

Eureka Server 2 配置

```
server.port=8082
eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=http://localhost:8082/eureka/
```

如果有多个 Eureka Server，则在注册时使用逗号（,）分隔，将除自身外的其他所有 Eureka Server 地址进行配置。

```
eureka.client.service-url.defaultZone=http://localhost:8081/eureka/,http://localhost:8082/eureka/
```

5.2 注册服务

Spring Cloud 基于 HTTP 协议的 RESTful 风格的 API 进行服务之间的通信，在注册服务时会明显地发现与编写 Spring Controller 没有太大区别。

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Discovery 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-service
version: 0.0.1-SNAPSHOT
packaging: jar
```

pom依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

② 在 Spring Boot 入口开启服务发现与注册支持。

```
@EnableDiscoveryClient
@SpringBootApplication
public class CloudServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(CloudServiceApplication.class, args);
    }
}
```

@EnableDiscoveryClient

开启 DiscoveryClient 的实例，与 Eureka Server 进行交互，负责注册服务、租约续期、检索服务、取消租约等功能。

③ 在 application.properties 配置 Eureka。

```
spring.application.name=SERVER-NAME
server.port=${PORT:${SERVER_PORT:0}}
eureka.client.service-url.defaultZone=http://localhost:8082/eureka
```

spring.application.name

应用名称，在服务调用时将使用此参数做依赖计算。

server.port

应用访问端口号。${PORT:${SERVER_PORT:0}} 表示自动分配一个未使用的端口，如果需要直接访问该服务则直接提供一个具体的端口号便可。

eureka.client.service-url

注册中心所在地址，如果 eureka 为集群，则通过逗号分隔配置所有的注册中心地址。

④ 编写服务。

与服务注册的代码 Spring MVC 的 Controller 并无区别，这样做极大地方便了对原先使用 Spring MVC 的项目进行微服务化重构的工作。

```
@RestController
public class ServerController {
    @RequestMapping(value = "hello")
    public String hello(@RequestParam("param") String param) {
        return "rpc: " + param;
    }
}
```

5.3 调用服务

各个微服务模块都是以 HTTP 协议暴露服务的，调用服务时只需使用类似 JDK 的 URLConnection 或 Spring 的 RestTemplate (HTTP 客户端) 便可，而 Spring Cloud Netflix 则提供了 Ribbon 与 Feign 工具来简化调用过程，并且支持客户端负载均衡、熔断等功能。

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Discovery 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-client
version: 0.0.1-SNAPSHOT
packaging: jar
```

每一个 Spring Cloud 应用都需要引入 Eureka Discovery 来接入到 Eureka 运行环境中，所以需要在 Spring Boot 的入口处通过 `@EnableDiscoveryClient` 注解添加发现服务能力，并且在 `application.properties` 中配置 Eureka 地址、应用名称、访问端口等基本信息。

5.3.1 Ribbon

Ribbon 是一个基于 HTTP 和 TCP 客户端的负载均衡器，通过客户端中配置的 ribbonServerList 服务端列表去轮询访问以达到均衡负载的作用。

Ribbon 核心组件

- Rule – 从服务列表中如何获取一个有效服务。
- Ping – 后台运行线程用来判断服务是否可用。
- ServerList – 服务列表。

② 在 `pom.xml` 文件中引入 ribbon 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

③ 在 `Spring Boot` 入口处注入 `RestTemplate` 实例。

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

RestTemplate

通过 Spring 自带的 Rest 客户端，可以方便地发起 HTTP 请求并且将结果序列化。

@LoadBalanced

该注解将开启 Ribbon 的负载均衡模式。开启后 Ribbon 将拦截 RestTemplate 发起的请求，并实现负

载均衡。

④ 调用服务。

```
@Autowired
private RestTemplate restTemplate;

public String say() {
    return restTemplate.getForObject("http://SERVER/hello?param=cloud", String.class);
}
```

http://SERVER/hello?param=cloud 是服务具体所在地址及传递的参数，与 URL 不同的是原本主机名 + 端口的地址变为服务名称 SERVER，由此可以推断出 Ribbon 客户端根据服务名称从 Eureka 注册中心寻找具体服务地址。在有多个服务提供者时由 Eureka 注册中心的服务列表与 Ribbon 配合完成负载均衡。

⑤ 在 application.properties 文件中配置 Ribbon。

当请求失败时，如下配置可以让 Ribbon 重试链接及更换其他服务提供方。

```
spring.cloud.loadbalancer.retry.enabled=true
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000
ribbon.ConnectTimeout=250
ribbon.ReadTimeout=1000
ribbon.OkToRetryOnAllOperations=true
ribbon.MaxAutoRetriesNextServer=2
ribbon.MaxAutoRetries=1
```

spring.cloud.loadbalancer.retry.enabled
是否开启重试机制，默认为关闭。

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds
断路器的超时时间，需要大于 ribbon 的超时时间，否则不会触发重试。

ribbon.ConnectTimeout
请求连接的超时时间。

ribbon.ReadTimeout
请求处理的超时时间。

ribbon.OkToRetryOnAllOperations
对所有操作请求都进行重试。

ribbon.MaxAutoRetriesNextServer

重试负载均衡其他实例的最大重试次数，不包括首次调用。

ribbon.MaxAutoRetries

同一个服务提供方最大重试次数，不包括首次调用。

以 ribbon 开头的配置项是针对所有服务的调用设置，如果需指定某一具体服务，只需在前面增加具体的应用名称，如：SERVER.ribbon.ConnectTimeout=250。

⑥ 配置 Ribbon 负载均衡策略。

```
@Configuration
public class RibbonConfiguration {

    @Bean
    public IRule ribbonRule() {
        return new BestAvailableRule();
    }

}
```

Ribbon 提供多种负载策略，由 IRule 进行管理。通过继承 `ClientConfigEnabledRoundRobinRule` 可自定义负载策略。

BestAvailableRule

最大可用策略，即先过滤出故障服务器，然后选择一个当前并发请求数最小的。

AvailabilityFilteringRule

可用过滤策略，先过滤出故障或并发请求大于阈值的一部分服务实例，然后再以线性轮询的方式从过滤后的实例清单中选出一个。

WeightedResponseTimeRule

带有加权的轮询策略，对各个服务器响应时间进行加权处理，然后再采用轮询的方式来获取相应的服务器。

RetryRule

在选定的负载均衡策略机上重试机制。

RoundRobinRule

以轮询的方式依次将请求调度不同的服务器（默认策略）。

RandomRule

随机选择一个服务提供方。

ZoneAvoidanceRule

区域感知轮询负载均衡。

使用 @RibbonClients 注解对所有服务的负载策略配置生效。

```
@EnableDiscoveryClient
@RibbonClients(defaultConfiguration = RibbonConfiguration.class)
@SpringBootApplication
public class CloudClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudClientApplication.class, args);
    }
}
```

defaultConfiguration 为负载策略配置文件。

如果希望针对某一具体服务配置负载策略，可以使用 @RibbonClient 注解单独配置。

```
@Configuration
@RibbonClient(name = "SERVER", configuration = RibbonConfiguration.class)
public class TestConfiguration {
}
```

name 为服务名称，configuration 为负载策略配置文件。

5.3.2 Feign

Feign 是一个声明式的 Web 服务客户端，通过简单的注解便可像调用本地方法一样调用远程服务。Spring Cloud 为 Feign 添加了 Spring MVC 的注解支持，并整合了 Ribbon 和 Eureka 来为 Feign 提供负载均衡功能。

② 在 pom.xml 文件中引入 Fegin 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

③ 在 Spring Boot 入口处开启 Fegin 支持。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class CloudClientApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(CloudClientApplication.class, args);
    }

}
```

@EnableFeignClients

将自动扫描所有 `@FeignClient` 注解。

④ 编写调用接口。

通过接口告知 Fegin 调用信息，定义的方法（`hello()`）则是来自于服务提供方暴露的具体服务方法（`RestController`）。

```
@FeignClient(name = "SERVER")
public interface ServerClient {
    @RequestMapping(value = "hello")
    public String hello(@RequestParam("param") String param);
}
```

name

服务提供方的应用名称。

url

手动指定调用地址。

decode404

是否开启 decoder 解码。为 true 时，当请求发生 404 错误会调用 decoder 进行解码；否则抛出 FeignException。

configuration

指定配置类，用于自定义 Encoder、Decoder、LogLevel、Contract。

path

定义当前 fegin 调用时的统一前缀。

fallback

指定容错处理类，当调用发生错误时会调用该指定的类。

fallbackFactory

指定容错处理的工厂类，用于减少重复的容错处理类编写。

⑤ 在 `application.properties` 中配置 Fegin。

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

feign.compression.request.enabled

开启请求压缩。

feign.compression.response.enabled

开启返回值压缩。

feign.compression.request.mime-types

设置压缩类型。

feign.compression.request.min-request-size

设置压缩最小阈值。

⑥ 调用服务。

```
@Autowired
private ServerClient serverClient;

public String say() {
    return serverClient.hello("cloud");
}
```

Spring Cloud 应用启动时，Feign 会扫描标有 `@FeignClient` 注解的接口生成代理，为每个接口方法创建一个 RequetTemplate 对象，并封装发起 http 请求时所需的所有信息，最终注册到 Spring 容器中。

5.4 Zuul网关

Spring Cloud 提供了 Zuul 作为服务网关，与 Dubbo 服务网关不同的是，Zuul 并不直接调用服务，而是通过动态路由提供代理服务，在具体开发过程中也更为简单方便。

① 新建 Spring Boot 工程，并在创建时勾选 `Eureka Discovery` 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-getway
version: 0.0.1-SNAPSHOT
packaging: jar
```

② 在 pom.xml 文件中引入 Zuul 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

③ 在 Spring Boot 入口处开启 Zuul 支持。

```
@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class CloudGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudGatewayApplication.class, args);
    }
}
```

EnableZuulProxy

设置一个 Zuul 的服务端点，并开启反向代理过滤器，以达到将请求转发到后端的服务提供方的目的。

④ 在 application.properties 配置 Zuul。

```
spring.application.name=geteway
server.port=8083
eureka.client.service-url.defaultZone=http://localhost:8082/eureka

zuul.routes.SERVER.path=/server/**
zuul.routes.SERVER.service-id=SERVER
```

zuul.routes.*.path

拦截请求的路径，其中星号（SERVER）为自定义的分组标记，用于路由配置。

zuul.routes.*.service-id

根据分组标记（SERVER）将拦截到的请求转发到某服务提供方的名称。

zuul.routes.*.url

根据分组标记（SERVER）将拦截到的请求转发到一个具体的 URL 地址。

⑤ 过滤器权限验证。

通过继承 ZuulFilter 类实现过滤功能，与 Spring MVC 的 HandlerInterceptor 类似。

```
public class RequestFilter extends ZuulFilter {

    @Override
```

```
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        String token = request.getParameter("token");
        if (token == null || !token.equals("1")) {
            ctx.setResponseBody("token error");
            ctx.setSendZuulResponse(false);
        }
        return null;
    }

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }

}
```

shouldFilter()

返回一个 Boolean 值，标识该过滤器是否开启。

filterType()

返回一个字符串，标识该过滤器在何时调用。

- `pre`：在请求被路由之前调用。
- `routing`：在路由请求时候被调用。
- `error`：处理请求时发生错误时被调用。
- `post`：在 routing 和 error 过滤器之后被调用。

filterOrder()

返回一个 int，标识该过滤器在过滤器链中执行的顺序。

run()

过滤器的具体逻辑。

`ctx.setResponseBody("token error");` 设置请求返回值。

`ctx.setSendZuulResponse(false);` 设置不对该请求进行路由操作。

当过滤器配置完成后，便可以在程序入口处注入该过滤器使他生效。

```
@Bean
public RequestFilter logFilter(){
    return new RequestFilter();
}
```

5.5 Hystrix 断路器

当远程请求失败时，Dubbo 通过 mock 实现服务降级与容错，而 Spring Cloud 则提供 Hystrix 达到同样的目的。与之不同的是 Hystrix 是以框架级别角度解决该问题，而 mock 则是以功能角度出发。

Hystrix 通过线程池来隔离资源，在使用时会根据调用的远程服务划分出多个线程池。例如调用产品服务的 Command 放入 A 线程池，调用账户服务的 Command 放入 B 线程池。当调用服务的代码存在 bug 或者由于其他原因导致自己所在线程池被耗尽时，不会对系统的其他服务造成影响。

Spring Cloud 提供的 Fegin 与 Ribbon 都非常好地支持了 Hystrix，只需简单的配置便可完成。

5.5.1 Ribbon

① 在 `pom.xml` 中引入 hystrix 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

② 在 `Spring Boot` 入口处开启 hystrix 支持。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class CloudClientApplication {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(CloudClientApplication.class, args);
    }
}
```

@EnableCircuitBreaker

开启熔断器。

③ 调用服务。

Ribbon 调用服务发生错误时，熔断机制生效，将调用由注解 @HystrixCommand 的 fallbackMethod 参数指定的方法，而不抛出服务异常，使整个系统保持正常运行。

```
    @HystrixCommand(fallbackMethod = "fallback")
    public String say() {
        return restTemplate.getForObject("http://SERVER/hello?param=cloud", String.
class);
    }

    public String fallback() {
        return "容错数据";
    }
```

fallbackMethod

指定一个处理回退逻辑的方法，回退方法应与需要熔断功能的方法具有相同返回值，并且要在同一个类中。

commandKey

该熔断器的名称，默认为 default。

defaultFallback

与 fallbackMethod 参数功能一致。

groupKey

服务所属分组

ignoreExceptions

调用时发生异常，则触发熔断机制，此参数可设置排除的异常。

threadPoolKey

为该熔断器分配一个自定义名称，配置资源隔离策略时使用。

threadPoolProperties

配置该熔断器的线程池参数。接受多个 @HystrixProperty 参数以 kv 形式配置。

示例：threadPoolProperties = {@HystrixProperty(name = "coreSize", value = "30") }

- coreSize：核心线程池大小和线程池最大值
- maxQueueSize：线程池队列最大值
- keepAliveTimeMinutes：线程池中空闲线程生存时间
- queueSizeRejectionThreshold：限定当前队列大小

④ 在 appaction.properties 配置熔断器。

```
spring.application.name=CLIENT
server.port=8889
eureka.client.service-url.defaultZone=http://localhost:8082/eureka

hystrix.command.default.execution.isolation.strategy=THREAD
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=1000
hystrix.command.default.execution.timeout.enabled=true
hystrix.command.default.execution.isolation.thread.interruptOnTimeout=true
hystrix.command.default.fallback.enabled=true
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=5000
hystrix.command.default.circuitBreaker.errorThresholdPercentage=50
hystrix.command.default.circuitBreaker.forceOpen=false
hystrix.command.default.circuitBreaker.forceClosed=false
hystrix.threadpool.default.coreSize=10
```

hystrix.command.*.execution.isolation.strategy

设置熔断器的资源隔离策略。

- THREAD：并发请求受线程池中的线程数量的限制（默认及推荐项）。
- SEMAPHORE：并发请求受到信号量计数的限制。

参数中 command 后的 default 为注解中 @HystrixCommand 的 commandKey 参数设置的值，默认为 default。

hystrix.command.*.execution.isolation.thread.timeoutInMilliseconds
超时时间。

hystrix.command.*.execution.timeout.enabled
是否开启超时。

hystrix.command.*.execution.isolation.thread.interruptOnTimeout
是否打开超时线程中断。

hystrix.command.*.fallback.enabled

设置 fallback 是否可用。

hystrix.command.*.circuitBreaker.sleepWindowInMilliseconds

触发熔断的时间间隔。

hystrix.command.*.circuitBreaker.errorThresholdPercentage

错误比率阀值，当错误超过或等于该值时，后续的请求将直接调用 fallback。

hystrix.command.*.circuitBreaker.forceOpen

强制打开熔断器，打开后将强制执行 fallback。

hystrix.command.*.circuitBreaker.forceClosed

强制关闭熔断器。

hystrix.threadpool.*.coreSize

核心线程池大小和线程池最大值。

与 `@HystrixCommand` 注解中的 `threadPoolProperties` 参数一致。

5.5.2 Fegin

Fegin 中已经集成了 Ribbon 与 Hystrix 依赖，在 `Dalston` 版本中默认并未开启，只需简单配置便可实现熔断功能。

① 编写回调类。

Fegin 通过接口确定调用信息，回调类只需实现接口便可，与 Dubbo 的 mock 类似，但不限制回调类的名称。

```
@Component
public class ServerClientFallback implements ServerClient {

    @Override
    public String hello(String param) {
        return "容错数据";
    }

}
```

另外还可以通过实现 `FallbackFactory` 接口，在 create 方法中返回服务接口的实例以实现回调。

```
@Component
public class DefaultFallback implements FallbackFactory<Object> {

    @Override
    public ServerClient create(Throwable cause) {
        return new ServerClient() {
            @Override
            public String hello(String param) {
                return "容错数据";
            }
        };
    }
}
```

② 配置回调类。

```
@FeignClient(name = "SERVER", fallback = ServerClientFallback.class,fallbackFactory
= DefaultFallback.class)
public interface ServerClient {

    @RequestMapping(value = "hello")
    public String hello(@RequestParam("param") String param);

}
```

fallback

指定具体的回调类，即当前接口的实现类。

fallbackFactory

指定回调工厂的实现类，即实现 FallbackFactory 接口的类。

③ 在 appaction.properties 中开启熔断器。

```
feign.hystrix.enabled=true
```

5.6 服务监控

将一个应用部署多次，以满足服务高可用需求，而监控每个应用的运行状态则是必不可少的事。因此 Spring Cloud 已经对每个应用的运行状态做了统计，并提供了简单的图形化界面管理工具。

同时活跃的 Spring 社区为 Spring Cloud 提供了 Spring-Cloud-Admin 集成管理工具，低侵入的方

式只需简单配置便可完成对分布式环境中各应用的监控。

主要功能如下：

- 显示应用信息
- 显示在线状态
- 日志级别管理
- JMX beans 管理
- 会话和线程管理
- 应用请求跟踪
- 应用运行参数信息
- 显示熔断器信息

spring cloud admin 可以通过 eureka 注册中心中的数据或专门提供的客户端获取应用信息。在 Spring Cloud 环境下 eureka 是最优选择，而客户端方式则常用于 Spring Boot 应用中。

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Discovery 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-admin
version: 0.0.1-SNAPSHOT
packaging: jar
```

② 在 pom.xml 文件中引入 admin 依赖。

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server</artifactId>
    <version>1.5.3</version>
</dependency>
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui</artifactId>
    <version>1.5.3</version>
</dependency>
```

③ 在 Spring Boot 入口处开启 admin 支持。

```
@SpringBootApplication
@EnableAdminServer
@EnableDiscoveryClient
public class CloudAdminApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudAdminApplication.class, args);
    }
}
```

@EnableAdminServer

标识当前应用是 Spring Cloud 监控应用。

④ 在 application.properties 文件中配置 admin。

```
spring.application.name=ADMIN
server.port=8899
eureka.client.service-url.defaultZone=http://localhost:8082/eureka
```

⑤ 配置客户端。

监控应用中的数据来自于 eureka 注册中心，所以要对所有 eureka 应用（在 Spring Boot 入口处添加过 @EnableDiscoveryClient 注解的应用）进行相应的配置。

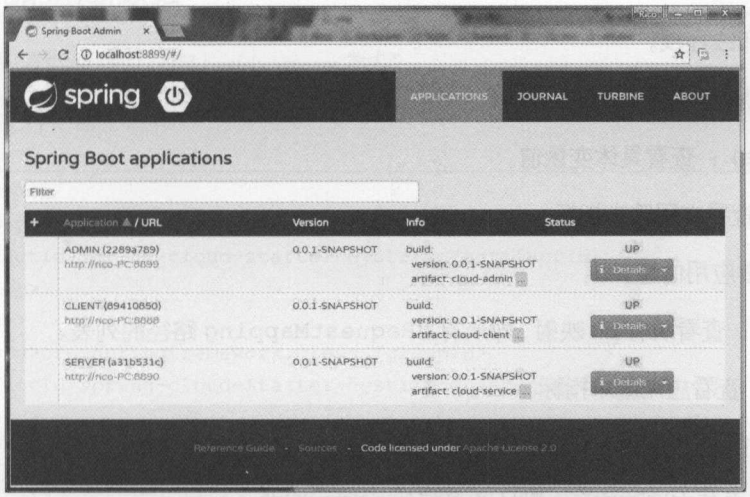
在各微服务应用的 pom.xml 文件中设置暴露应用基本信息。

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <executions>
                <execution>
                    <goals>
                        <goal>build-info</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

⑥ 测试。

依次启动注册中心、cloud-service、cloud-client、cloud-admin 后，用浏览器访问 http://localhost:8899 便可看见 Spring-Cloud-Admin 的管理界面。

虽然 admin 从注册中心的列表中获取已注册应用，但目前仅展示了应用的基础信息，需要继续对 eureka 中的各个应用进行配置以获取应用的运行时及熔断器数据。



5.7 应用监控

① 在各微服务模块中的 pom.xml 文件中添加监控依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
</dependency>
```

应用健康信息

Actuator 是 Spring Boot 提供的监控应用系统数据和管理生产环境的模块，可以使用 HTTP、JMX、

SSH、Telnet 等方式管理和监控应用。

以 HTTP 为例，可以通过浏览器轻松访问如下信息。

- `/autoconfig`：查看自动配置的使用情况。
- `/configprops`：查看配置属性，包括默认配置。
- `/beans`：bean 及其关系列表，显示一个应用中所有 Spring Beans 的完整列表。
- `/dump`：打印线程栈。
- `/env`：查看所有环境变量。
- `/env/{name}`：查看具体变量值。
- `/health`：查看应用健康指标。
- `/info`：查看应用信息。
- `/mappings`：查看所有 url 映射，即所有 `@RequestMapping` 路径的列表。
- `/metrics`：查看应用基本指标。
- `/shutdown`：关闭应用。
- `/trace`：查看基本追踪信息，默认为最新的 HTTP 请求。

JMX Beans

Jolokia 是一个 JMX-HTTP 桥，它提供了一种访问 JMX beans 的替代方法。为 `Spring-Cloud-Admin` 提供 beans 相关信息。

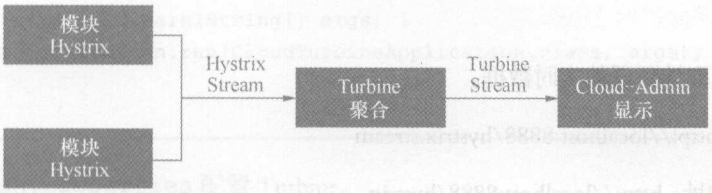
② 在 `application.properties` 文件中配置权限。

```
management.security.enabled=false
```

actuator 中暴露了很多应用的敏感信息，所以默认进行了权限限制，通过 `security.enabled` 将其关闭方便配置。

5.8 熔断器监控

Hystrix 记录了触发熔断时的数据，并提供了简单的图形化统计面板，Hystrix 的数据来自各个应用，只能分别查看，管理起来比较麻烦，为了简化 Spring Cloud 的管理，提供了 Turbine 来聚合所有的 Hystrix，同时 Spring Cloud Admin 也很好地支持了 Turbine，只需简单的配置便可集成。



5.8.1 单应用的熔断数据

在每一个应用中配置熔断器以获取 hystrix.stream 数据。

① 在 pom.xml 文件中添加依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

Fegin 中虽然已集成了 Hystrix，但为了让监控数据生效还需再引用。

spring-cloud-starter-hystrix-dashboard

熔断器监控面板依赖。

② 在 Spring Boot 入口处开启 Hystrix 面板。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableHystrix
@EnableHystrixDashboard
public class CloudClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(CloudClientApplication.class, args);
    }
}
```

通过 @EnableHystrixDashboard 开启 Hystrix 面板的支持，如果只想使用 Spring Cloud Admin 的 Turbine 展示数据，则可以不用依赖 spring-cloud-starter-hystrix-dashboard。

③ 测试。

启动应用后便可获取熔断器的实时数据。

hystrix 数据地址：http://localhost:8888/hystrix.stream

hystrix dashboard 地址：http://localhost:8888/hystrix

hystrix.stream

熔断器的实时数据，ping 数据为空时，访问一个使用了熔断器的服务便可。

hystrix dashboard

将 `hystrix.stream` 地址提供给它，便可解析成可视化图表。启动应用后便可获得聚合了指定应用的 hystrix.stream。

5.8.2 使用Turbine聚合数据

Turbine 负责整合所有应用的 hystrix.stream 数据，为了分担压力所以将 Turbine 独立成一个新的应用。

① 新建 Spring Boot 工程，并在创建时勾选 `Eureka Discovery` 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-turbine
version: 0.0.1-SNAPSHOT
packaging: jar
```

② 在 `pom.xml` 添加 Turbine 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
```

③ 在 `Spring Boot` 入口处开启 Turbine。

```
@EnableTurbine
@SpringBootApplication
public class CloudTurbineApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(CloudTurbineApplication.class, args);
    }
}
```

④ 在 application.properties 配置 Turbine。

```
spring.application.name=TURBINE
server.port=8892
eureka.client.service-url.defaultZone=http://localhost:8082/eureka
management.security.enabled=false
turbine.app-config=CLIENT
turbine.aggregator.clusterConfig=default
turbine.clusterNameExpression=new String("default")
```

turbine.app-config

指定要聚合熔断数据的应用名称，多个应用以逗号分隔。

turbine.aggregator.clusterConfig

指定需要聚合的集群名称。

turbine.clusterNameExpression

获取集群名表达式。

⑤ 测试。

启动应用后便可获得聚合了指定应用的 hystrix.stream。

通过 http://localhost:8892/turbine.stream 地址便可访问所有的应用熔断数据。

```
: ping
data: {"reportingHostsLast10Seconds":1,"name":"meta","type":"meta","timesta
mp":1503519020000}
```

将 turbine.stream 地址传送给 hystrix dashboard 便可以解析多个熔断监控图表。

5.8.3 Cloud Admin整合Turbine

在 cloud-admin 应用中继续改造以让它能够解析 Turbine 的图表，完成统一监控。

① 在 pom.xml 中添加 Turbine 依赖。

```
<dependency>
    <groupId>de.codecentric</groupId>
```

```
        <artifactId>spring-boot-admin-server-ui-turbine</artifactId>
        <version>1.5.3</version>
    </dependency>
```

② 在 application.properties 中配置 Turbine。

```
spring.boot.admin.turbine.clusters=default
spring.boot.admin.turbine.location=TURBINE
```

spring.boot.admin.turbine.location

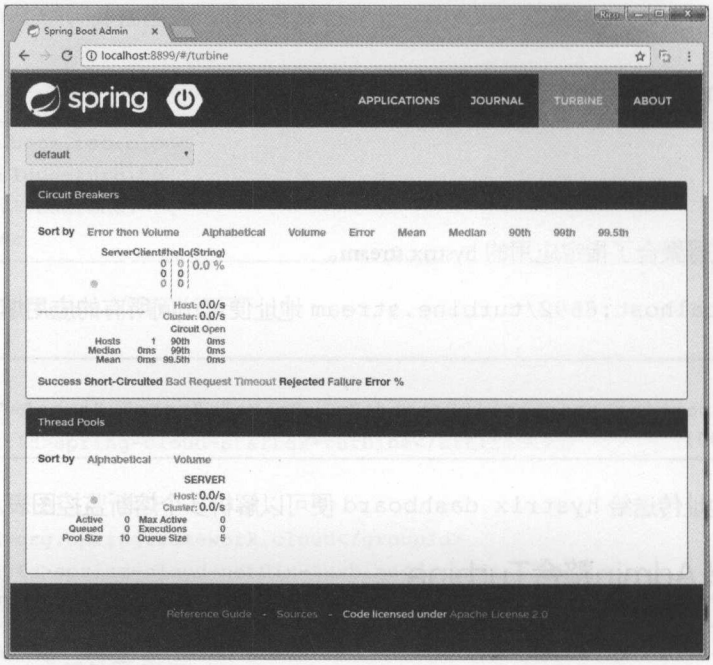
turbine 应用的名称或地址，它将自动寻找并解析 turbine.stream。

spring.boot.admin.turbine.clusters

聚合的集群名称。

③ 测试。

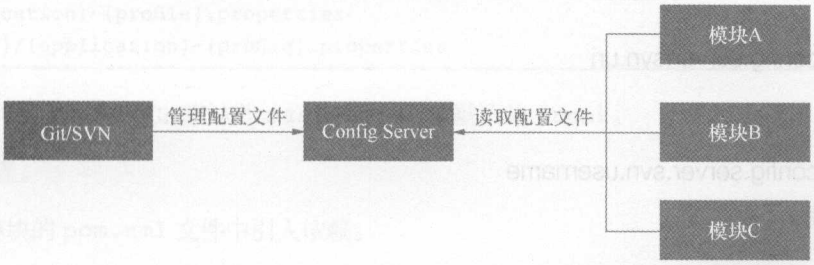
启动应用后便可在 cloud-admin 的 Turbine 目录看到各个应用的统计图表。



5.9　统一管理配置文件

在工作中一般会进行开发环境、测试环境、生产环境的区分，不同的环境由不同的配置文件管理，当

拆分的模块数量较多时，无疑众多的配置文件会显得混乱并且不便于管理。为此 Spring Cloud 提供了 spring-cloud-config-server 结合 SVN 或 Git 对这些配置文件进行集中管理。



① 新建 Spring Boot 工程，并在创建时勾选 Eureka Discovery 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-config
version: 0.0.1-SNAPSHOT
packaging: jar
```

② 在 pom.xml 文件中引入 Config 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
    <groupId>org.tmatesoft.svnkit</groupId>
    <artifactId>svnkit</artifactId>
</dependency>
```

svnkit

SVN 客户端支持，此例使用 SVN 作为配置文件管理工具，如果使用 Git，则不必引入该依赖。

③ 在 application.properties 配置 Config Server。

```
spring.application.name=CONFIG
server.port=8894
eureka.client.service-url.defaultZone=http://localhost:8082/eureka
spring.profiles.active=subversion
spring.cloud.config.server.svn.uri=https://localhost:443/svn/config
spring.cloud.config.server.svn.username=username
spring.cloud.config.server.svn.password=password
```

spring.profiles.active

指定当前配置文件管理工具，如果为 Git 可不写，则将后续配置中的 server.svn 变为 server.git 便可完成对 Git 工具的配置。

spring.cloud.config.server.svn.uri

svn 仓库地址。

spring.cloud.config.server.svn.username

svn 访问用户名。

spring.cloud.config.server.svn.password

svn 访问密码。

④ 上传配置文件到 SVN。

```
// 文件名: 'client-dev.properties'
config=hello config
```

⑤ 测试。

启动工程后用浏览器访问 http://localhost:8894/client/dev/ 便可获得 client-dev.properties 的配置文件内容。

```
{
    name: "client",
    profiles: [
    "dev"
    ],
    label: null,
    version: null,
    state: null,
    propertySources: [{
    name: "https://localhost:443/svn/config/trunk/client-dev.properties",
    source: {
                config: "hello config"
            }
        }
    ]
}
```

不难发现，访问的 url 路径与配置文件名存在一定的映射关系。

配置文件名称格式为 {application}-{profile}.properties，对应 url 如下：

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

其中 label 为分支名称，Git 默认为 master 而 SVN 默认为 trunk。

⑥ 应用端配置。

在各微服务模块的 pom.xml 文件中引入依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

在 src/main/resources 目录中新建 bootstrap.properties 并配置 Config Server。

```
spring.cloud.config.name=client
spring.cloud.config.profile=dev
spring.cloud.config.label=trunk
spring.cloud.config.uri=http://localhost:8894/
```

spring.cloud.config.name

对应 SVN 配置文件名称中 {application} 部分。

spring.cloud.config.profile

对应 SVN 配置文件名称中 {profile} 部分。

spring.cloud.config.label

对应的分支。

spring.cloud.config.uri

Config Server 地址。

直接提供 config server 的访问地址意味着普通的 Sring Boot 应用也可以使用 Config Server 配置，如果基于 Eureka 注册中心使用的话，需要配置 eureka.client.service-url.defaultZone 参数以提供注册中心的地址与 spring.cloud.config.discovery.service-id 参数提供 config server 的应用名称。

基于 Spring Boot 的 Dubbo 应用也可以使用。

本章示例代码详见异步社区网站本书页面。

第6章　数据持久化

6.1　Spring Data MySQL

6.2　Spring Data MongoDB

6.3　Spring Data ElasticSearch

6.4　TCC 分布式事务

6.5　Spring Data Redis

企业级应用绝大多数都是围绕着数据库的 CURD 操作进行的，得益于 Spring Boot Starter 对常用数据库的封装，可以非常方便且快速与其集成，这里将介绍最为常用的 3 种数据库。

为了确保各微服务保持自身的独立性及整个分布式架构的效率，在设计服务模块时应尽量保持每个微服务模块使用单一且独立的数据源，各微服务模块之间的数据库互不干扰。

6.1 Spring Data MySQL

Spring Data 基于 Spring 提供了统一编程模型，并且支持众多不同的数据库，在保证底层数据特性的前提下，为关系型数据库或非关系型数据库提供了统一的操作方式，极大地简化了开发与学习难度。

6.1.1 依赖与配置

① 在 pom.xml 文件中添加依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

spring-boot-starter-data-jpa

Spring Data JPA 是 Spring 基于 ORM 框架、JPA 规范的基础上封装的一套 JPA 应用框架，可使开发者用极简的代码即可实现对数据的访问和操作。它提供了包括增、删、改、查等在内的常用功能，且易于扩展。

> JPA (Java Persistence API) 是 Sun 官方提出的 Java 持久化规范。
>
> 它提供了一种对象 / 关联映射工具来管理 Java 应用中的关系数据。

mysql-connector-java

MySQL 核心依赖。

② 在 application.properties 文件中配置数据源。

```
spring.datasource.url=jdbc:mysql://domain:3306/test?useUnicode=true&characterEncoding=utf8
spring.datasource.username=xxxx
spring.datasource.password=xxxx
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.jpa.database=MYSQL
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

spring.datasource.url

数据源地址。

spring.datasource.username

用户名。

spring.datasource.password

密码。

spring.datasource.driverClassName

启动名称。

spring.jpa.database

指定使用的数据库。

spring.jpa.show-sql

控制台中显示执行的 SQL。

spring.jpa.hibernate.ddl-auto

自动创建表时所采用的策略。

- create：每次加载hibernate时都会删除上一次生成的表，然后根据你的model类再重新生成新表，即使两次没有任何改变也要这样执行。
- create-drop：每次加载 hibernate 时根据 model 类生成表，sessionFactory 一但关闭，表就自动删除。
- update：第一次加载 hibernate 时根据 model 类会自动建立起表的结构，以后加载 hibernate 时根据 model 类自动更新表结构，即使表结构改变了表中的行也仍然存在，不会删除以前的行。
- validate：每次加载 hibernate 时，验证创建数据库表结构，只会和数据库中的表进行比较，不会创建新表但是会插入新值。

6.1.2 实体映射

Spring Boot 应用在启动时链接 MySQL 数据库，并根据程序中的实体及配置的策略自动在数据库中创建相应的表及外键关系。

```
@Entity
public class Bag {

    @Id
    private long id;

    private String name;
```

```
    // getter and setter
}

@Entity
public class Curriculum {

    @Id
    private long id;

    private String name;

    // getter and setter
}

@Entity
public class School {

    @Id
    private long id;

    private String name;

    // getter and setter
}

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(nullable = false, unique = true)
    private String name;

    @ManyToOne
    private School school;

    @OneToMany
    private Set<Curriculum> curriculum;

    @OneToOne
    private Bag bag;

    // getter and setter
}
```

@Entity

标识该类为实体类，其中的 name 参数用于指定数据库中的表名。默认以类名作为表名。

@Id

标识当前字段为主键。

@GeneratedValue

指定主键的自增长策略。

- AUTO：主键由程序控制。
- IDENTITY：主键由数据库控制。
- SEQUENCE：根据底层数据库的序列来生成主键，条件是数据库支持序列。
- TABLE：使用一个特定的数据库表格来保存主键。

@Column

标识实体类中属性与数据表中字段的对应关系。

- name：数据库表中对应字段的名称。
- unique：唯一标识。
- nullable：表示该字段是否可以为 null 值。
- insertable：在使用 INSERT 脚本插入数据时，是否需要插入该字段的值。
- updatable：在使用 UPDATE 脚本插入数据时，是否需要更新该字段的值。
- columnDefinition：创建表时，该字段创建的 SQL 语句。
- table：包含当前字段的表名。
- length：当字段的类型为 varchar 时，指定字段的长度。
- precision：数值的总长度。
- scale：小数点所占的位数。

@ManyToOne

指定与 SchoolEntity 实体（表）的关系为多对一，即多个学生属于一个学校。

- targetEntity：指定具体实体。
- cascade：指定级联关系策略。
  - CascadeType.REFRESH：获取数据库中的最新数据。
  - CascadeType.PERSIST：同步新增。
  - CascadeType.MERGE：同步更新。
  - CascadeType.REMOVE：同步删除。
  - CascadeType.ALL：以上策略总和。
- fetch：控制加载数据策略。

- FetchType.EAGER：查询到父实体类的时候加载。
- FetchType.LAZY：第一次访问数据库的时候加载。

- optional：指定是否为必须。

@OneToMany

指定与 CurriculumEntity 实体（表）的关系为一对多，即一个学生有多个课程。

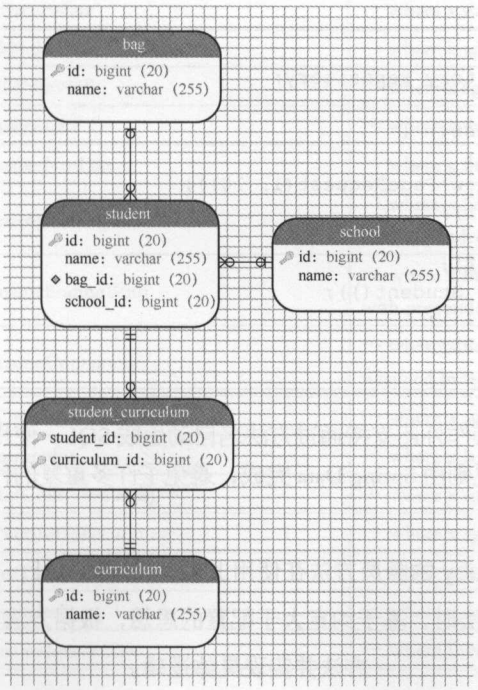
通过注解 @JoinColumn 可以在 curriculum 表中新增一个外键字段，以确保两表之间的关系，否则 JPA 会新增一个多余的关联表。

- orphanRemoval：是否开启自动删除外键为 null 的数据。
- mappedBy：双向关联实体时，指定两者谁是维护端。
- cascade：与 ManyToOne 的一致。
- fetch：与 ManyToOne 的一致。

@OneToOne

指定与 BagEntity 实体（表）的关系为一对一，即一个学生有一个书包。

最终根据实体所建立的表与主外键关系如图所示：



6.1.3 Repository

Spring Data JPA 默认预先生成了一些基本的 CURD（增、删、查）方法，创建接口并继承相应的 Repository（资源库）便可获得数据库的 DAO 操作功能，并被 Spring 容器加载。

> Repository：通过用来访问领域对象的一个类似集合的接口，在领域与数据映射层之间进行协调，也就是 DAO。

- `Repository`：标识任何继承它的均为仓库接口类，方便 Spring 自动扫描识别。
- `CrudRepository`：继承自 Repository，实现了一组 CRUD 相关的方法。
- `PagingAndSortingRepository`：继承自 CrudRepository，实现了一组分页排序相关的方法。
- `JpaRepository`：继承自 PagingAndSortingRepository，实现一组 JPA 规范相关的方法。

```
public interface StudentRepository extends PagingAndSortingRepository<Student, Long> {

}
```

PagingAndSortingRepository

继承 `Repository` 需要操作的实体类及 id 的数据类型以便完成映射。

6.1.3.1 简单查询

```
@Autowired
private StudentRepository repository;

public void studentTest() {
  repository.findAll();
  repository.findAll(new PageRequest(1, 10));
  repository.findOne(1L);
  repository.count();
  repository.exists(1L);
  repository.save(new Student());
  repository.deleteAll();
  repository.delete(1L);
}
```

Spring Data 除了提供基本的操作外还支持通过方法名自动生成 SQL，使用时只需根据约定好的规则定义方法名，而方法的具体实现则由 Spring Data 完成，避免了许多重复的代码，让开发者有更多的精力聚焦在业务逻辑上。

> 动态语言 Ruby 的幽灵方法在使用上也有相同的特征，在调用一个不存在的方法时，基于约定大于配置的思想，根据方法名动态生成方法并实现具体逻辑。

定义方法名为：

```
List<Student> findByNameLikeOrderByIdAsc(String name);
```

将自动生成 SQL：

```
select * from student s where s.name like ?1 order by s.id asc;
```

这里 ?1 为传参方式，表示方法中的第一个参数（String name），如果选择第二个参数则为 ?2。

关键词、使用方法、生成SQL规则对照表

关键词	方法名	SQL
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is、Equals	findByFirstnameIs,findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull、NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection age)	… where x.age not in ?1
TRUE	findByActiveTrue()	… where x.active = true
FALSE	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

6.1.3.2 分页查询

PagingAndSortingRepository 从名称中可看出已经集成了分页与排序功能，在使用时只需在定义的方法中传入 Pageable 参数并设置 Page 作为返回值。

① 编写接口。

```
    Page<Student> findAllBy(Pageable pageable);
```

② 查询使用。

```
    Page<Student> result = repository.findAll(new PageRequest(1, 10, new Sort(Sort.Direction.
DESC, "id")));
    List<Student> content = result.getContent();
    for (Student student : content) {
        System.out.println(student.getName());
    }
    int count = result.getTotalPages();
    System.out.println(count);
```

PageRequest

Pageable 的实现类，封装了分页所需的参数，如当前页、每页大小、排序等参数。

Page

封装了查询结果集，通过 getContent() 获得具体查询内容，getTotalPages() 获得总页数。

6.1.3.3 自定义SQL

Spring Data 可以通过 @Query 注解自定义 SQL 语句以支持更复杂的查询、删除、更新操作，但不支持保存操作。

```
    @Query(value = "SELECT * FROM student s WHERE s.id <= ?1", nativeQuery = true)
        List<Student> myQuery(long id);

        @Query("select s from Student s where s.id <= ?1")
        List<Student> myQuery2(long id);

        @Query(value = "select * from student \n#pageable\n ", countQuery = "SELECT COUNT(*)
FROM student", nativeQuery = true)
        Page<Student> myQueryAll(Pageable pageable);

        @Modifying
        @Query("update Student s set s.name = :name where s.id = :id")
        void myUpdate(@Param("name") String name, @Param("id") int id);

        @Modifying
        @Query(value = "delete from Student WHERE id = :id", nativeQuery = true)
        void myDelete(@Param("id") long id);
```

@Query

- value：SQL 语句。
- nativeQuery：是否开启原生 SQL，为 false 时则使用 JPQL 作为查询语言，SQL 中的表名需要改为实体名以支持映射。
- countQuery：分页时用于统计总数的 SQL。

> JPQL 是 Java 持久性查询语言，专门为 Java 应用程序访问和导航实体实例设计，主要用于处理 JPA 实体。

@Modifying

告知 Spring Data 该 SQL 语句为 update 或 delete。

传参

将参数传递给 SQL 语句有两种方式。

- SQL 语句中的 ?1 将按方法中参数的位置获取参数，如果为第二位则为 ?2。
- @Param("name") 定义参数名称，在 SQL 语句中通过 :name 获取。

分页

对方法进行分页配置后，@Query 注解中的 countQuery 参数为必要项，并通过 \n#pageable\n 将分页参数传递给 SQL 语句。

6.1.4 JdbcTemplate

```
    @Autowired
    private JdbcTemplate jdbcTemplate;

    jdbcTemplate.execute("drop table if exists account");
    jdbcTemplate.execute("create table account(id int , nickname varchar(255))");
    jdbcTemplate.batchUpdate("insert into account(id,nickname) values(?,?)", Arrays.
asList(new Object[]{1, "昵称111"}, new Object[]{2, "昵称222"}));
    jdbcTemplate.update("update account set nickname = ? where  id = ?", "昵称", 1);
    List result = jdbcTemplate.queryForList("select * from account where nickname
like ?", "昵称");
    jdbcTemplate.batchUpdate("update account set nickname  = ? where id = ?;", Arrays.
asList(new Object[]{"批量修改111", 1}, new Object[]{"批量修改222", 2}));
```

execute()

执行 SQL 语句。

update()

更新数据，并返回所影响的行数。

batchUpdate()

根据传入的数组大小批量执行 SQL 语句，并返回每次执行的 SQL 所影响行数的 int 数组。

queryForList()

根据条件查询数据并返回多个结果，同样还可以使用 queryForObject() 查询并只返回一个结果。

6.1.5　事务管理

一个完善的业务逻辑通常需要操作多张数据表，当业务逻辑执行到一半发生异常，导致业务逻辑无法完成时，为了确保数据的完整性需要将以前的已操作的数据进行回滚处理，在引入 JDBC 或 JPA 依赖时 Spring Boot 已经默认注入了事务管理的实例，在使用时只需在需要开启事务的方法或者类上添加 @Transactional 注解便可。

```
    @Transactional(rollbackFor = RuntimeException.class, propagation = Propagation.
REQUIRED, isolation = Isolation.DEFAULT)
    public void demo() {
        //操作数据的业务逻辑
    }
```

rollbackFor

指定该事物针对什么异常进行回滚，可以接受多个异常名称。

propagation

指定该事物的传播策略。

- REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
- REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。
- NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 REQUIRED。

isolation

指定该事务的隔离级别，具体如下：

- DEFAULT：使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是：READ_COMMITTED。
- READ_UNCOMMITTED：一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。
- READ_COMMITTED：一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。
- REPEATABLE_READ：一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。
- SERIALIZABLE：所有的事务依次逐个执行，事务之间完全不会产生干扰。

6.2 Spring Data MongoDB

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似 JSON 的 BJSON 格式，因此可以存储比较复杂的数据类型。Mongo 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

传统的关系数据库一般由数据库（database）、表（table）、记录（record）三个层次概念组成，MongoDB 是由数据库（database）、集合（collection）、文档对象（document）三个层次组成。MongoDB 对应关系型数据库里的表，但是集合中没有列、行和关系概念，这体现了模式自由的特点。

MongoDB 中的一条记录就是一个文档，是一个数据结构，由字段和值对组成。适合对大量或者无固定格式的数据进行存储，比如：日志、缓存等。对事务支持较弱，不适用复杂的多文档（多表）的级联查询。

6.2.1 依赖与配置

① 在 pom.xml 中添加依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

② 在 application.properties 文件中配置 MongoDB。

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=book
```

MongoDB 3.0版本以下配置

- spring.data.mongodb.host：连接地址。
- spring.data.mongodb.port：端口号。
- spring.data.mongodb.database：数据库名称。

MongoDB 3.0版本以上配置

- spring.data.mongodb.uri：连接配置。

单MongoDB配置

```
mongodb://name:pass@host:port/database
```

集群以逗号分隔配置

```
mongodb://name:pass@host1:port1,host2:port2/database
```

6.2.2 实体映射

```
@Document
public class Article {

    @Id
    private ObjectId id;

    @Indexed
    @Field("nickname")
    private String name;

    // getter and setter
}
```

@Document

标识当前类为 MongoDB 的集合，基于 Spring Data 的规范，该实体类可以与 Spring Data MySQL

的 @Entity 混合使用。

- collection：设定该集合的别名，默认为实体类名。

@Id

文档对象中的 _id 字段是必要的，默认情况下 Spring Data 会自动匹配实体类中名为 id 的变量，并自动映射到文档中的 _id 字段。通过该注解可以将实体类中其他非 id 的变量与文档中的 _id 字段建立映射关系。

在新增数据未设置 id 参数数据时，ObjectId 将产生一个不重复的随机数作为文档对象 _id 字段的值，同样如果当实体中未指定 id 变量，Spring Data 默认以 ObjectId 类型映射到 _id 字段。

@Field

设定该字段在 mongodb 中的别名。

@Transient

默认情况下所有的私有字段都映射到文档，该注解标识的字段会从存储在数据库中的字段列中被排除（即该字段不保存到 mongodb）。

@Indexed

声明该字段需要索引，建索引可以大大地提高查询效率。

@CompoundIndex

复合索引的声明，建复合索引可以有效地提高多字段的查询效率。

6.2.3 Repository

与操作 MySQL 时的 Repository 一样，新建接口继承 MongoRepository 并提供实体类与 id 的类型便可。MongoRepository 继承自熟悉的 PagingAndSortingRepository，可以看出 MySQL 对表的操作方式同样适用于对 MongoDB 文档的操作。

```
public interface ArticleRepository extends MongoRepository<Article, ObjectId> {

}
```

自定义查询

```
@Query(value = "{'name': ?0}", fields = "{'name':1}")
Page<Article> myQuery(String name, Pageable pageable);
```

通过 @Query 注解可以使用 MongoDB 原生查询语言。分页与传参方式和 MySQL 的 Query 注解一致。

value

查询语句。

fields

指定要返回的列。

count

标识该方法的作用是否为统计总数。

delete

标识该方法的作用是否为删除数据。

exists

标识该方法的作用为判断数据是否存在。

6.2.4 MongoTemplate

MongoTemplate 实现了 MongoOperations 与 ApplicationContextAware 接口，基于 Spring 容器提供了一组基本的 MongoDB 的操作方法，只需通过 Spring 的 @Autowired 注解注入便可使用。

```
@Autowired
private MongoTemplate mongoTemplate;

public void mongodbDemo() {
    Query query = new Query();
    Criteria criteria = Criteria.where("name").is("book");
    query.addCriteria(criteria);
    Pageable pageable = new PageRequest(0, 2, Direction.ASC, "name");
    query.with(pageable);
    List<ArticleEntity> results = mongoTemplate.find(query, Article.class);
    for (Article article : results) {
        System.out.println(article.getName());
    }
```

Criteria

Criteria 是标准查询的接口，引用静态的 Criteria.where 把多个条件组合在一起（链式编程），这样就可以轻松地将多个方法标准和查询连接起来，方便操作查询语句。

链式编程：调用的每个方法都返回该方法的对象引用，因而这个引用可以产生下一个方法调用，最终生成一个调用链。

Pageable

封装了分页所需的参数。

Query

构建查询语句。

6.3 Spring Data ElasticSearch

ElasticSearch 是一个高可扩展的开源全文搜索和分析引擎，它允许存储、搜索和分析大量的数据，并且这个过程是近实时的。它通常被用作底层引擎和技术，为复杂的搜索功能和要求提供动力。`Spring Data Elasticsearch` 封装了 Elasticsearch 客户端的 API 接口，提供了一套与 MySQL、MongoDB 极其接近的操作方式，极大地降低了学习成本。

当用户搜索文章时，直接将关键词提交给数据库做模糊查询（`where title like "%keyword%"`）显然不是好的做法，这样做不单效率低下并且搜索关键词也不够精确。借助 Elasticsearch 解决方案，可以对文章建立索引并将搜索关键词进行分词处理，通过被拆分出的多个关键词搜索文章索引，最终将查询出的结果集与关键词进行相关度排序。

6.3.1 基本概念

索引（Index）

一个索引包含许多特征类似的文档。例如，有一个标识用来标识用户数据，另一个索引用来标识产品目录，其他的索引可以标识其他数据。一个索引需要指定一个名称（必须全部小写），执行索引、搜索、修改和删除操作，需要指定对应的索引名称。

在一个集群中，你可以创建多个索引。

类型（Type）

在一个索引中，可以定义多个类型。一个类型可以管理索引中符合特定逻辑的一部分数据。一般来说，类型定义具有公共字段的文档。例如你想创建一个博客平台，并且使用一个索引存储所有数据。在这个索引中，你可以定义一个类型用来存储用户数据，另一个类型用来存储博客数据，还可以创建一个类型用来存储评论。

文档（Document）

文档是能够被索引的基础单元。例如一个文档存储一个用户信息，另一个文档存储一个产品信息。Elasticsearch 文档使用 JSON(JavaScript Object Notation) 来表现数据。

在一个索引 / 文档中可以存储许多文档。需要注意，尽管一个文档在物理上是被存储在一个索引中的，但文档必须被索引或分配给索引的类型。

映射（mapping）

映射类似于关系型数据库中的模式 (schema) 定义。每个索引都存在一个映射，它定义了该索引中的每一种类型，以及索引相关的配置。映射可以显式定义，或者在文档被索引时自动创建。在实现上，它定义外部数据转化到 ES 内部数据的流程，由一个或者多个 analyzer 组成，每个 analyzer 又由多个 filter 组成。

Elasticserach层级

要想理解 Elasticsearch 中索引、类型、文档、域（字段），可以和关系型数据库做个简单的对比。

- 关系型数据库：数据库 → 表格 → 行 → 列
- Elasticsearch：索引 → 类型 → 文档 → 字段

6.3.2 安装与运行

Spring Data Elasticsearch 与 Elasticsearch 存在版本依赖关系，为了避免不必要的麻烦，尽量选择 RELEASE 版本。因此选择 2.4.x 版本的 elasticsearch 提供全文检索服务。

Spring data elasticsearch	Elasticsearch
3.0.0.RC2	5.5.0
3.0.0.M4	5.4.0
2.0.4.RELEASE	2.4.0
2.0.0.RELEASE	2.2.0
1.4.0.M1	1.7.3
1.3.0.RELEASE	1.5.2
1.2.0.RELEASE	1.4.4
1.1.0.RELEASE	1.3.2
1.0.0.RELEASE	1.1.1

选择 2.4.6 版本的 elasticsearch 下载后解压，Windows 用户运行 bin 目录下的 `elasticsearch.bat`，Linux/Mac 用户运行 `elasticsearch.in.sh` 便可启动。

启动后通过浏览器运行 http://localhost:9200 访问 Elasticsearch 可得到如下信息：

```
{
    "name": "Ludi",
    "cluster_name": "elasticsearch",
    "cluster_uuid": "ocT8ZhvWT06mPT4klTsurw",
    "version": {
        "number": "2.4.6",
        "build_hash": "fcbb46dfd45562a9cf00c604b30849a6dec6b017",
        "build_timestamp": "2017-01-03T11:33:16Z",
        "build_snapshot": false,
        "lucene_version": "5.5.2"
    },
    "tagline": "You Know, for Search"
}
```

6.3.3 基于HTTP交互

```
curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING> '-d <BODY>
```

- VERB：HTTP 方法，GET、POST、PUT、HEAD、DELETE。
- PROTOCOL：HTTP 协议或 HTTPS 协议。
- HOST：请求的节点主机。
- PORT：端口号，默认 9200。
- PATH：请求路径。
- QUERY_STRING：可选请求参数。
- BODY：JSON 形式请求的主题。

索引文档

```
curl -X<PUT> 'http://localhost:9200/index/type/id'-d <BODY>
```

建立索引时必须要指定文档所属索引与类型并指定 id，如果 id 未指定的话 elasticsearch 会随机分配一个。文档的具体内容则在 PUT 请求的 Body 部分以 JSON 格式提交。

请求示例：

```
PUT http://localhost:9200/book/article/1
```

Body 参数：

```
{
    "name":"Spring Data Elasticsearch"
}
```

返回结果如下所示：

```
{
    "_index": "book",
    "_type": "article",
    "_id": "1",
    "_version": 1,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    },
    "created": true
}
```

获取索引

```
curl -X<GET> 'http://localhost:9200/index/type/id'-d
```

获取索引时必须指定索引与类型并且指定 id，以 GET 提交请求。

请求示例：

```
GET http://localhost:9200/book/article/1
```

返回结果如下所示：

```
{
    "_index": "book",
    "_type": "article",
    "_id": "1",
    "_version": 1,
    "found": true,
    "_source": {
        "name": "Spring Data Elasticsearch"
    }
}
```

删除索引

```
curl -X<DELETE> 'http://localhost:9200/index/type/id'-d
```

删除索引时指定索引、类型、id 后删除具体 id 下的文档，如果只指定索引，则删除索引下的所有文档，以 DELETE 提交请求。

请求示例：

```
DELETE http://localhost:9200/book/article/1
```

返回结果如下所示：

```
{
    "found": true,
    "_index": "book",
    "_type": "article",
    "_id": "1",
    "_version": 2,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    }
}
```

修改文档

```
curl -X<POST> 'http://localhost:9200/index/type/id'-d <BODY>
```

修改文档与创建索引一致，根据指定的 id，如果已存在则覆盖修改，PUT 与 POST 请求方式均可。当修改成功后会影响 _version 参数加 1。

请求示例：

```
POST http://localhost:9200/book/article/1
```

Body 参数：

```
{
    "name":"修改后的内容"
}
```

返回结果如下所示：

```
{
    "_index": "book",
    "_type": "article",
    "_id": "1",
    "_version": 13,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    },
    "created": false
}
```

搜索索引

```
curl -X<GET> 'http://localhost:9200/index/_search?q=keyword'-d
```

指定索引后通过访问 _search 开始搜索，通过 q 传递搜索关键词，以 GET 发起请求。

请求示例：

```
GET http://localhost:9200/book/_search?q=修改后的内容
```

返回结果如下所示：

```
{
    "took": 5,
    "timed_out": false,
    "_shards": {
        "total": 5,
        "successful": 5,
        "failed": 0
    },
    "hits": {
        "total": 1,
        "max_score": 0.30018792,
        "hits": [
            {
                "_index": "book",
                "_type": "article",
```

```
                "_id": "1",
                "_score": 0.30018792,
                "_source": {
                    "name": "修改后的内容"
                }
            }
        ]
    }
}
```

- hits：搜索结果。
- hits.total：符合搜索条件的总数。
- hits.max_score：最大匹配相关度分数。
- hits.hits：实际搜索结果数据。

6.3.4 配置分词器

在 Elasticsearch 中，索引分析模块可以通过注册分词器 (Analyzer) 来进行配置。分词器的作用是当一个文档被索引的时候，它能从文档中提取出若干词元 (Token) 来支持索引的存储和搜索。

例如关键词为：我爱你中国，经过分词器处理后会切分成：我、爱你、中国。

Elasticsearch 拥有许多分词器、包含内置的和第三方的，这里主要介绍 IK Analysis for Elasticsearch 分词器来解决中文分词的需求。

IK Analysis 与 Elasticsearch 版本存在一定关系，并且基于 Spring Data Elasticsearch 的版本支持，最终选择使用 2.4.5 版本的 Elasticsearch 与 1.10.5 版本的 IK Analysis。

将下载好的 IK Analysis 插件解压后，移入 Elasticsearch 的 plugins 目录下（如果没有 plugins 目录可自行创建），重启 Elasticsearch 后便已经加载了 IK Analysis 分词器。

分词演示

```
POST http://localhost:9200/_analyze?analyzer=ik
```

Body 参数：

```
中国崛起
```

返回结果如下所示：

```
{
  "tokens": [{
        "token": "中国崛起",
        "start_offset": 0,
        "end_offset": 4,
        "type": "CN_WORD",
        "position": 0
    },{
        "token": "中国",
        "start_offset": 0,
        "end_offset": 2,
        "type": "CN_WORD",
        "position": 1
    },{
        "token": "崛起",
        "start_offset": 2,
        "end_offset": 4,
        "type": "CN_WORD",
        "position": 2
    }]
}
```

- start_offset：起始位置。
- end_offset：结束位置。
- type：文字类型，CN_WORD 表示中文。
- position：词元所在原始关键词的位置。

快速示例

① 创建索引。

```
PUT http://localhost:9200/book
```

② 创建映射。

```
POST http://localhost:9200/book/article/_mapping
```

Body 参数：

```
{
  "properties": {
```

```
    "content": {
        "type": "string",
        "analyzer": "ik_max_word",
        "search_analyzer": "ik_max_word"
    }
  }
}
```

通过 _mapping 指定字段 content 使用 ik_max_word 作为分词器。

③ 创建文档。

```
POST http://localhost:9200/book/article/1
{"content":"下饭萝卜条，到底怎样腌制的呢？"}
POST http://localhost:9200/book/article/2
{"content":"那些扬名国外的中国美食"}
POST http://localhost:9200/book/article/3
{"content":"那些诱人的中韩美食！"}
```

④ 搜索。

```
GET http://localhost:9200/book/article/_search?q=中韩
```

分词器将关键词“中韩”切分成：“中”“韩”两个词元进行搜索。

返回结果如下所示：

```
{
  "took": 6,
  "timed_out": false,
  "_shards": {
      "total": 5,
      "successful": 5,
      "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 0.11506979,
    "hits": [{
        "_index": "book",
        "_type": "article",
        "_id": "3",
        "_score": 0.11506979,
        "_source": {
            "content": "那些诱人的中韩美食！"
```

```
            }
        },{
            "_index": "book",
            "_type": "article",
            "_id": "2",
            "_score": 0.008439008,
            "_source": {
                "content": "那些扬名国外的中国美食"
            }
        }
    ]
  }
}
```

6.3.5 依赖与配置

① 在 pom.xml 中引入 elasticsearch 依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
</dependency>
```

② 在 application.properties 中配置 elasticsearch。

```
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

spring.data.elasticsearch.cluster-nodes

elasticsearch 节点所在地址，集群环境中多个节点以逗号分隔

端口需要指定 9300，而非 HTTP 访问的 9200。

spring.data.elasticsearch.cluster-name

节点名称，默认为 elasticsearch。

spring.data.elasticsearch.repositories.enabled

是否开启仓库中数据存储，默认为 true。

6.3.6 实体映射

```
@Document(indexName = "book", type = "article")
public class Article {
```

```
    @Id
    private int id;

    @Field(type = FieldType.String, index = FieldIndex.analyzed, store = true, analyzer
= "ik_max_word", searchAnalyzer = "ik_max_word")
    private String content;

    // getter and setter
}
```

@Document

建立实体与数据映射关系。

- indexName：索引名称，可以理解为数据库名。
- type：类型名称，可以理解为表名。

@Field

修饰文档字段。

- type：字段类型，未指定则使用该参数的类型。
- index：是否开启分词器。
- store：是否开启排序。
- analyzer：指定分词器。
- searchAnalyzer：指定搜索时的分词器。

6.3.7 Repository

与操作 MySQL 时的 Repository 一样，新建接口继承 ElasticsearchRepository，提供实体类与 id 的类型便可。

ElasticsearchRepository 继承自 ElasticsearchCrudRepository，而 ElasticsearchCrudRepository 则继承自 PagingAndSortingRepository，可以看出 ElasticsearchRepository 拥有操作 elasticsearch 数据及分页等功能。

```
public interface ArticleRepository extends ElasticsearchRepository<Article,
Integer> {

}
```

可以看见 ElasticsearchRepository 中提供了常用 CURD 操作，与 MySQL、MongoDB 并无区别。与之不同的是，它专门服务于搜索数据的 search 方法。

```
    @Autowired
    private ArticleRepository repository;

    QueryBuilder query = QueryBuilders.queryStringQuery("中").field("content").
analyzer("ik_max_word");
    Page<Article> results = repository.search(query, new PageRequest(0, 100));
    for (Article article : results) {
        System.out.println(article.getContent());
    }
```

通过 QueryBuilder 构建查询，QueryBuilders 封装查询条件中的具体细节。它根据不同的场景提供了用于字符串类型字段的 QueryStringQueryBuilder 构造器，和用于 elasticsearch 原生查询语句的 ScriptQueryBuilder 构造器。

- queryStringQuery：搜索关键词。
- field：限定搜索的字段，可以指定多个，如果不指定，则进行全文检索。
- analyzer：指定分词器，如果实体中已指定，则可以忽略。

6.3.8 ElasticsearchTemplate

```
    @Autowired
    private ElasticsearchTemplate elasticsearchTemplate;

    SearchQuery search = new NativeSearchQueryBuilder()
        .withQuery(QueryBuilders.queryStringQuery("中"))
        .withFilter(QueryBuilders.queryStringQuery("国"))
        .withPageable(new PageRequest(0, 10))
        .withIndices("book", "book2")
        .build();
    AggregatedPage<Article> results = elasticsearchTemplate.queryForPage(search,
Article.class, new SearchResultMapper() {
        @Override
        public <T> AggregatedPage<T> mapResults(SearchResponse response, Class<T> clazz,
Pageable pageable) {
            List<Article> content = new ArrayList<>();
            for (SearchHit hit : response.getHits()) {
                Article entity = new Article();
                // 可以在这里过滤搜索到的结果
                entity.setContent(hit.getSource().get("content").toString());
```

```
                content.add(entity);
            }
            return new AggregatedPageImpl<>((List<T>) content, pageable, response.getHits().
getHits().length);
        }
    });

    for (Article article : results) {
        System.out.println(article.getContent());
    }
```

在 elasticsearchTemplate 提供的搜索方法中，如果需要 SearchResultMapper 接口为参数，则表示可以拦截搜索结果，并且在实现类中处理结果。上述代码只实现了取出结果，并返回了简单逻辑。

SearchQuery

构建搜索请求。

- withQuery：查询语句与之前的 QueryStringQueryBuilder 一致。
- withFilter：过滤条件。
- withPageable：分页参数。
- withIndices：指定的搜索索引，如果未指定则只在指定的实体中检索。

SearchResultMapper

在构建搜索请求的时候可以指定多个索引，并且搜索时也是对全文进行检索，所以数据结构的差异化必然存在，通过 SearchResultMapper 重写 mapResults 方法可以对异构的搜索结果进行梳理，形成统一的数据结构对外输出。

- response：搜索结果，包含 elasticsearch 相关信息。
- clazz：调用方法时传入的实体映射类。
- pageable：分页参数。
- AggregatedPage：封装了结果集，继承自 Page<T> 接口，与分页参数 Pageable 配合完成分页操作。

将 response 的结果序列化成 JSON 后发现，其与使用 HTTP 的 _search 接口查询到的结果一致，实际搜索到的结果在 hits 节点下，通过 hit.getSource() 获取到 Map 结构的结果集合中，key 部分为字段，value 部分为具体结果。

6.4　TCC 分布式事务

在单应用中操作数据库时，一般会开启事务以确保数据的一致性。但在分布式架构中基于远程调用进行业务处理时，传统事务管理机制就无法发挥作用。此时则可以使用 TCC（Try-Confirm-Cancel）二阶事务管理策略。

> 例如现在有订单与钱包两个模块，产生订单时需要在钱包中扣除订单相应的金额，但是在订单模块中调用了钱包扣除金额的接口后发生了异常，对订单操作的数据在本地事务的管理下已经进行了回滚操作，但由于这两个模块拥有不同的运行环境，进行远程调用时并不被同一个事务管理，所以钱包模块中已操作的数据无法进行回滚。

初步操作（Try）

初步操作是整个 TCC 的核心部分，此阶段对资源进行操作但并不直接生效。整个操作过程由本地事务管理，从 TCC 整体看，可以理解为 Try 阶段 + 本地事务共同构建了分布式事务管理。

确认操作（Confirm）

当业务操作全部完成并没有错误发生后，将 Try 阶段已经操作但并不生效的数据进行提交确认处理，使其生效。

取消操作（Cancel）

在执行业务操作时，如果发生错误抛出异常，则对 Try 阶段已经操作的数据进行回滚。

使用 Dubbo 和 Cloud 进行远程调用时，目标服务的异常信息将会传递给调用方，从而触发调用方的本地事务。基于这一特性下面演示 TCC 机制的简单实现。

① 在服务提供方编写 TCC 逻辑。

```
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional
    public void serviceTry(){
        WalletEntity entity =  jdbcTemplate.queryForObject("select * from wallet_entity
where id = ?", new Object[]{1}, WalletEntity.class);

        if(entity.getState().equals("trying")){
            throw new RuntimeException("资源占用中");
        }
```

```
        jdbcTemplate.update("update wallet_entity set amount = amount - ? , state = 'trying'
where id = ?", 5, 1);
    }

    @Transactional
    public void serviceConfirm(){
        jdbcTemplate.update("update wallet_entity set  state = 'confirm' where id = ?", 1);
    }

    @Transactional
    public void serviceCancel(){
        jdbcTemplate.update("update wallet_entity set amount = amount + ? , state = 'cancel'
where id = ?", 5, 1);
    }
```

serviceTry()

在用户钱包表余额中减去相应的金额，并通过 state 字段对其加锁，标识该资源在申请状态中。

serviceConfirm()

正式提交操作，并解锁资源。

serviceCancel()

回滚在 Try 阶段已操作的数据，并解锁资源。

② 在服务调用方执行 TCC 逻辑。

```
    @Autowired
    private ServiceClient serviceClient;

    try {
        serviceClient.serviceTry();
        serviceClient.serviceConfirm();
    } catch (Exception e) {
        serviceClient.serviceCancel();
    }
```

一个完善的业务逻辑通常会远程调用多个服务，所以编写业务逻辑时需要基于每个服务的 try 方法进行，当所有的业务逻辑编写完成后，再统一调用每个服务的 confirm 方法提交。当某一个远程调用的服务返回了异常信息时，则需要在 catch 块中调用每个服务的 cancel 方法回滚之前的操作。

ServiceClient 用于调用远程方法，如 Spring Cloud 中的 Fegin

6.5 Spring Data Redis

Redis 是一款开源的高级键值 (key-value) 缓存 (cache) 和存储 (store) 系统，单线程运行、拥有较高的读写能力，可以用作网络化的内存缓存。

> 为了获得更高的 IO 性能，Redis 启动时会将所有数据加载到内存中，而备份时才会将数据持久化到硬盘中。由此可见，Redis 无法替代 MySQL 或 MongoDB 来作为数据库被使用，而适合于为高频读写的数据提供缓存服务。

在分布式环境下非常适合为各个微服务模块提供公共数据共享服务，并且为单应用模块提供缓存服务。

`spring-boot-starter-data-redis` 封装的 `Jedis`（Redis 的 Java 客户端）拥有丰富的 Redis 操作方式，并且针对热数据提供了缓存解决方案，只需简单的配置便可实现开箱即用。

6.5.1 安装运行

Linux下安装

将下载好的安装包解压，进入目录后使用 `make` 命令编译 Redis。

编译好后进入 `src` 目录，通过命令 `./redis-server ../redis.conf &` 启动。

- `redis-server`：redis 服务端应用入口。
- `redis.conf`：配置文件，如果不提供，则使用默认配置。
- `&`：linx 命令，标识在后台运行。

Windows下安装

双击下载好的 msi 格式安装包进行安装，如果是 zip 格式的安装包只需解压。

解压完成后进入 Redis 所在的目录，双击 `redis-server.exe` 或使用 cmd 命令 `redis-server.exe redis.windows.conf` 启动服务。

- `redis-server.exe`：redis 服务端应用入口。
- `redis.windows.conf`：配置文件，如果不提供则使用默认配置。

常用命令

Redis 提供了许多命令来完成相应的操作，启动 Redis 服务端后，Windows 平台启动 `reds-cli.exe`，Linux 启动 `./redis-cli` 进入命令行交互模式。

命令	描述
PING	检测 Redis 服务是否启动
EXISTS key	判断一个 key 是否存在
SET key value	设置 key 的值为 value，如果 key 存在，则覆盖更新
SETNX key value	设置 key 的值为 value，如果 key 存在，则不做任何操作
SETEX key seconds value	设置 key 的值为 value，并指定 seconds 时间后过期（删除），单位为秒
APPEND key value	当该 key 的值类型为字符串时，向尾部追加指定的 value
GET key	获取 key 的值
KEYS pattern	返回满足条件的所有 key 值，可以是具体 key 名或者 *（星号）通配查询
DEL key	删除一个 key
TYPE key	返回值的类型
EXPIRE key seconds	设置 key 在 seconds 时间后过期
TTL key	查看 key 的过期时间
PERSIST key	取消 key 的过期时间
QUIT	退出命令行交互模式

6.5.2 依赖与配置

① 在 `pom.xml` 文件中添加依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

② 在 appaction.properties 文件中配置。

```
spring.redis.database=0
spring.redis.host=localhost
spring.redis.pool.max-active=-1
spring.redis.pool.max-idle=-1
spring.redis.pool.max-wait=-1
spring.redis.pool.min-idle=-1
spring.redis.port=6379
spring.redis.timeout=-1
```

spring.redis.database

指定链接的数据库，Redis 指定了编号为 0-15 的 16 个数据库。

spring.redis.host

服务端所在地址。

spring.redis.port

服务端所使用端口，默认为 6379。

spring.redis.pool.max-active

指定连接池最大的活跃连接数，-1 表示无限，默认为 8。

spring.redis.pool.max-idle

指定连接池最大的空闲连接数，-1 表示无限，默认为 8。

spring.redis.pool.max-wait

指定当连接池耗尽时，重新获取连接需要等待的最大时间，以毫秒单位，-1 表示无限等待。

spring.redis.pool.min-idle

指定连接池中空闲连接的最小数量，默认为 0。

spring.redis.timeout

指定连接超时时间，毫秒单位，-1 表示无限，默认为 0。

6.5.3 缓存支持

提供的服务接口在处理业务逻辑时会涉及到数值计算或者查询数据库等行为，对服务器造成一定的压力，当请求量大了之后这种差异会尤为明显。

服务接口在处理完业务逻辑后最终产出结果，如果已知该结果在短时间内不会发生变化，那么将结果

存入读写性能较高的 Redis 中，下次再请求该接口时直接从 Redis 读取计算好的结果，不再执行真实的业务逻辑，从而达到优化性能的目的。

这一缓存逻辑 Spring Boot 已经实现并且封装好，只需简单的配置及 `@Cacheable` 注解便可使用。

① 在 `pom.xml` 中添加缓存依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

② 在 `appaction.properties` 中指定缓存提供者。

```
spring.cache.type=redis
```

指定缓存提供者，如果未指定，则会根据下面的顺序去检测：

- Generic
- JCache (JSR-107)
- EhCache 2.x
- Hazelcast
- Infinispan
- Redis
- Guava
- Simple

③ 在 `spring boot` 入口处开启缓存支持。

```
@SpringBootApplication
@EnableCaching
public class SpringBootRedisApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootRedisApplication.class, args);
    }
}
```

@EnableCaching

开启缓存支持，并且自动配置 CacheManager（缓存管理器）。

④ 为方法添加注解。

```
@Component
@CacheConfig(cacheNames = "CacheService")
public class CacheService {

  @Cacheable(condition = "#id < 10")
  public long cache(int id) {
        return System.currentTimeMillis();
    }
}
```

为方便测试，cache() 方法返回了当前时间戳，首次调用时将当前时间戳缓存进 Redis 中，再次调用时直接从 Redis 取得该方法的返回结果，时间戳将不会改变，由此判断缓存机制是否生效。

@Cacheable

应用于方法上，标识该方法开启缓存支持。

- key：缓存对象存储在 Map 集合中的 key 值，非必需，默认将函数的所有参数组合作为 key 值，可以使用 SpEL 表达式配置。例如：key = "#p0" 表示使用函数第一个参数作为缓存的 key 值。
- value：指定缓存存储的集合名，必须值。
- cacheNames：等同于 value 参数。
- condition：设置缓存规则，引用 SpEL 表达式，在方法调用前根据传入参数判断是否缓存。例如：#id < 10 表示当该方法的 id 参数小于 10 时才进行缓存。
- unless：设置缓存规则，引用 SpEL 表达式，在方法调用后根据返回结果判断是否缓存。例如：#result > 10000 表示当返回值大于 10000 时才进行缓存。
- sync：是否开启同步缓存，在多线程环境中，高并发情况下缓存的数据将会被多个线程计算多次，通过设置该属性为 true 告诉 redis cache lock cache entry，这样就只有 1 个线程计算。
- keyGenerator：指定缓存的 key 生成策略，与 key 参数互斥。
- cacheManager：指定缓存策略。
- cacheResolver：指定缓存解析器。

@CachePut

应用于方法上，能够根据参数定义条件来进行缓存，与 @Cacheable 不同的是，它每次都会真实调用函数，所以主要用于数据新增和修改操作。它的参数与 @Cacheable 类似，具体功能可参考上面

对 @Cacheable 参数的解析。

@CacheEvict

应用于方法上，通常用在删除方法上，用来从缓存中删除相应数据。除了同 @Cacheable 一样的参数之外，它还有下面两个参数：

- allEntries：是否删除所有缓存的数据。
- beforeInvocation：在调用方法之后删除缓存的数据，设置为 true 时，在调用方法之前移除数据。

@CacheConfig

应用于类之上，对该类中所有开启了缓存注解的方法进行统一参数配置。@Cacheable 注解中的 value 参数为必须项，可以在该注解中配置 cacheNames 参数以起到减少配置的作用。

⑤ 配置缓存策略。

调用 CacheService 类的 cache() 方法后，在 Redis 命令行交互模式下使用 KEYS * 便可查看到刚刚缓存的数据。但是根据 Spring Cache 提供的缓存策略，该缓存数据会一直存在，在实际项目中如果不处理好缓存数据删除机制，则很容易产生脏数据，让系统变得复杂起来。为此 Spring Cache 提供了 CachingConfigurerSupport 类来扩展自定义的缓存策略。

```
@Configuration
public class RedisCacheConfig extends CachingConfigurerSupport {

    @Bean
    public CacheManager cacheManager(RedisTemplate redisTemplate) {
        RedisCacheManager manager = new RedisCacheManager(redisTemplate);
        manager.setDefaultExpiration(100);
        return manager;
    }

    @Bean
    public KeyGenerator keyGenerator() {
        return new KeyGenerator() {
            @Override
            public Object generate(Object target, Method method, Object... params) {
                StringBuilder sb = new StringBuilder();
                sb.append(target.getClass().getName());
                sb.append(method.getName());
                for (Object obj : params) {
                    sb.append(obj.toString());
                }
                return sb.toString();
            }
```

```
        };
    }

}
```

CacheManager

指定 Redis 缓存管理器作为当前缓存策略，通过 `manager.setDefaultExpiration(100)` 方法可为所有的缓存设置默认过期时间，单位为秒。

KeyGenerator

配置缓存 key 生产策略为：当前缓存方法所在的类名 + 方法名 + 参数。

6.5.4　RedisTemplate

Redis 不适合用于数据持久化，所以 Spring Data 也未提供与 MySQL 或 MongoDB 等类似的 `repository` 接口，而只提供了封装了 Jedis 客户端的操作模板。

```
@Resource
private RedisTemplate<String, Object> redisTemplate;

public void redis() {
  ValueOperations<String, Object> operations = redisTemplate.opsForValue();
  //存入redis
  operations.set("key", "value");
  //存入redis时设置超时时间
  operations.set("key2", "value", 100, TimeUnit.MINUTES);
  //根据key获取value
  Object key = operations.get("key");
  //根据key设置超时时间
  redisTemplate.expire("key", 10, TimeUnit.SECONDS);
  //根据key删除数据
  redisTemplate.delete("key2");
  //查询所有的key
  Set<String> keys = redisTemplate.keys("*");
}
```

Redis 在存储数据前会先将内容进行序列化处理，所以 `RedisTemplate<K,V>` 需要分别指定 key 与 value 对应的数据类型，同时提供以 String 类型序列化的 `StringRedisTemplate` 模板。

根据 Redis 支持的数据结构，提供以下模板：

- `opsForValue`：键值类型数据操作。

- opsForList：数组类型操作。
- opsForSet：集合类型操作。
- opsForZSet：有序集合操作。
- opsForHash：哈希结构操作。

向 Redis 存入数据后会发现，key 与 value 都进行了十六进制处理，虽然节省存储空间但是很难阅读。为了在开发过程中方便调试，可以配置 RedisTemplate 将 key 与 value 序列化成 JSON 格式。

① 在 pom.xml 中引入 fastjson。

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.38</version>
</dependency>
```

fastjson 是阿里巴巴开源高效率的对象与 JSON 序列化、反序列化工具。

② 在 RedisCacheConfig 配置类中注入 RedisTemplate 配置。

```
@Bean
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory connectionFactory) {
  RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();
  template.setConnectionFactory(connectionFactory);
  GenericFastJsonRedisSerializer fastJsonRedisSerializer = new GenericFastJsonRedisSerializer();
  template.setDefaultSerializer(fastJsonRedisSerializer);
  return template;
}
```

GenericFastJsonRedisSerializer 类实现了 RedisSerializer 接口，为 RedisTemplate 提供序列化方案。

- setDefaultSerializer()：指定默认的序列化方案，包含 key 与 value 部分的序列化。
- setValueSerializer()：指定 value 部分的序列化方案。
- setKeySerializer：指定 key 部分的序列化方案。

6.5.5 全局锁

Redis 可以非常好地为各个微服务应用提供一个公共的数据交换空间，但当多个客户端（微服务应用）同时访问一个公共数据时，难免会互相竞争导致混乱。

为了避免这一情况发生，程序在访问数据之前先获取一个全局锁，以确保该数据在这一段时间内只允许有一个应用在操作，当操作完成后再释放锁。

Redis 的 setnx 命令天生适合用来实现锁的功能，这个命令只有在键不存在的情况下才为键设置值。获取锁之后，其他程序再设置值就会失败，即获取不到锁。获取锁失败，只需不断地尝试获取锁，直到成功获取锁，或者到设置的超时时间为止。另外为了防治死锁，即某个程序获取锁之后，程序出错而没有释放，其他程序无法获取锁，从而导致整个分布式系统无法获取锁以致引起一系列问题，甚至导致系统无法正常运行。这时需要给锁设置一个超时时间，即 setex 命令，锁超时后，其他程序就可以获取锁了。

① 编写全局锁逻辑。

```
@Component
public class DistributedLockHandler {

    public final static long LOCK_EXPIRE = 30 * 1000L;
    private final static long LOCK_TRY_INTERVAL = 30L;
    private final static long LOCK_TRY_TIMEOUT = 20 * 1000L;

    @Autowired
    private RedisTemplate redisTemplate;

    public boolean getLock(String key, String value) {
        try {
            if (StringUtils.isEmpty(key) || StringUtils.isEmpty(value)) {
                return false;
            }
            long startTime = System.currentTimeMillis();
            do {
                if (!redisTemplate.hasKey(key)) {
                    redisTemplate.opsForValue().set(key, value, LOCK_EXPIRE, TimeUnit.
MILLISECONDS);
                    return true;
                }
                if (System.currentTimeMillis() - startTime > LOCK_TRY_TIMEOUT) {
                    return false;
                }
                Thread.sleep(LOCK_TRY_INTERVAL);
            } while (redisTemplate.hasKey(key));
        } catch (InterruptedException e) {
            e.printStackTrace();
            return false;
        }
        return false;
```

```
        }

        public void releaseLock(String key) {
            if (!StringUtils.isEmpty(key)) {
                redisTemplate.delete(key);
            }
        }
    }
```

getLock() 方法主要处理锁的逻辑，首先根据锁的 key 值判断是否存在，如果不存在则返回 ture，并设置一个带有时效性的锁；当检测到锁存在时表示当前时间段内已加锁，无法继续操作，等待一段时间后再次判断 key 是否存在，在规定的时间内检测到锁一直存在，则直接返回 false，表示当时资源已被锁住。

releaseLock() 释放掉指定的锁。

② 测试。

```
    try {
        // 初次尝试并未锁住，将获得到锁
        System.out.println(distributedLockHandler.getLock("key", "value") ? "获得到锁" :
"被锁住");
        // 再次尝试发现已经锁住
        System.out.println(distributedLockHandler.getLock("key", "value") ? "获得到锁" :
"被锁住");
        // 模拟锁超时
        Thread.sleep(DistributedLockHandler.LOCK_EXPIRE);
        // 再次请求锁已超时，将再次获得到锁
        System.out.println(distributedLockHandler.getLock("key", "value") ? "获得到锁" :
"被锁住");
        // 手动释放锁
        distributedLockHandler.releaseLock("key");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

在进行业务逻辑处理前通过 getLock() 方法获取到一个锁，设定当前资源被锁住，其他客户端（微服务应用）无法进行操作，处理完成后再通过 releaseLock() 释放掉该锁。

通过以上代码可以看出，第一次执行获取锁时，将打印获得到锁，而第二次再获取时该资源已被锁住。休眠一段时间后将超过该锁的有效时间，自动释放掉后再次调用，成功获得锁，最终通过 releaseLock() 将锁释放掉。

本章示例代码详见异步社区网站本书页面。

第7章　表单验证

服务模块在处理业务逻辑之前，通常需要对传入的参数进行验证，Spring Mvc 配合 Hibernate 提供的 Validator 模块只需在传入参数实体中对各个参数使用注解便可完成复杂的验证工作。

Dubbo 框架允许在基于 Spring MVC 实现的网关模块中对传入参数进行验证，验证成功后再调用相应的服务模块。

Spring Cloud 中的各个模块都是基于 HTTP 协议进行通信的，默认情况下每个模块都集成了 Spring MVC 部分，所以可以在每个模块中验证参数。

① 在 pom.xml 文件中引入依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

② 编写验证实体。

```
public class AccountValidate {

    @NotBlank(message = "用户名不能为空", groups = {PUSH.class, MODIFY.class})
    @Pattern(regexp = "^[a-z0-9_-]{3,15}$", message = "用户名格式错误")
    private String name;

    @NotNull(message = "年龄不能为空", groups = PUSH.class)
    @Min(value = 18, message = "最小年龄为18")
    private Integer age;

    @DecimalMax(value = "100", message = "金额不能大于100")
    private Double price;

    @AssertFalse(message = "不能为false")
```

```
    private boolean male;

    @DateTimeFormat(pattern = "yyyy-MM-dd hh:mm:ss")
    @Past(message = "应该提供之前的日期")
    private Date birthday;

    @Email(message = "email格式错误")
    private String email;

    @Size(min = 10, max = 1000, message = "内容长度应该在10至1000个字之间")
    private String content;
    public interface PUSH {}

    public interface MODIFY {}

  // getter and setter
}
```

注解描述

注解	作用
AssertTrue	值为 true
AssertFalse	值为 false
Null	引用为空
NotNull	引用不为空
NotEmpty	字符串引用和值都不为空
NotBlank	字符串引用和值都不为空
Max	必须大于指定的数值，可以用来验证 int 或 long 类型数值
Min	必须小于指定的数值，可以用来验证 int 或 long 类型数值
DecimalMax	必须大于指定的数值，可以用来验证 Double 或 Float 类型数值
DecimalMin	必须小于指定的数值，可以用来验证 Double 或 Float 类型数值
Past	必须是过去的时间，@DateTimeFormat 注解用来格式化时间
Future	必须是未来的时间
Pattern	字符串必须匹配正则表达式
Size	字符串的长度必须在指定的范围内
Email	字符串必须是一个有效的电子邮箱
URL	字符串必须是一个有效的 URL
Valid	级联检查关联的实体

③ 编写 Controller 并接收参数。

```
    @RequestMapping(value = "/", method = RequestMethod.POST)
    public Object test(@Validated({AccountValidate.PUSH.class, AccountValidate.MODIFY.
class}) AccountValidate validate, BindingResult result, HttpServletResponse response) {
      if (result.hasErrors()) {
        Map<String, String> errors = new HashMap<>();
        for (FieldError error : result.getFieldErrors()) {
          errors.put(error.getField(), error.getDefaultMessage());
        }
        response.setStatus(500);
        return errors;
      } else {
        return "success";
      }
    }
```

将接收参数的 AccountPOJO 实体作为形参时，Spring MVC 会将实体与 URL 请求中的传参建立映射关系，同时增加 @Validated 注解表示开启参数验证。每一个验证实体的后面都必须要紧跟着一个 BindingResult 参数用来接收验证的结果。

其中，@Validated 所跟的 PUSH.class 与 Modify.class 用于区分当前验证组，即在 AccountValidate 类中修饰字段验证格式时的 group 配置的属性，如果添加则表示该 controller 所接收的传入参数值只验证指定范围内的字段，如果不传入则表示验证全部。

以上代码通过 result.hasErrors() 判断验证结果是否有错误，并且通过 result.getFieldErrors() 获取每一个字段对应的错误信息，最终改变 response 状态码并返回错误信息。

本章示例代码详见异步社区网站本书页面。

第8章　定时任务

8.1　Spring Task 单机定时任务

8.2　Cron 表达式

8.3　QuartZ 分布式定时任务

在实际项目中经常会需要在具体的时间点执行某程序，如每周报表统计等。Spring Boot 提供了 Sping Task 来解决单机情况下的定时任务需求，在分布式架构中，则可以使用 QuartZ 来协调调度多个定时任务从而避免重复计算的问题。

8.1　Spring Task 单机定时任务

Spring Boot 已默认集成了 Spring Task，无需任何配置通过注解便可直接使用。

```
@Component
@EnableScheduling
public class BookTask {

    private long count = 0;

    @Scheduled(fixedRate = 1000)
    public void taskA() {
        System.out.println("当前时间: " + System.currentTimeMillis() + "\n执行次数: " + count++);
    }
}
```

@EnableScheduling

扫描所有的 Scheduled 注解，使定时任务生效，该注解也可以配置于 Spring Boot 入口处。

@Scheduled

配置于方法上，根据设定的规则将定时调用该方法。

- fixedRate：指定定时任务的循环间隔时间，以时间点为判断标准。
- fixedDelay：指定定时任务的循环间隔时间，以方法执行结束后为判断标准。
- initialDelay：初始化定时任务的延迟执行时间。
- zone：指定 cron 表达式所处的时区，如果为空，则使用当前机器的默认时区。
- cron：指定 cron 表达式以支持更为丰富的定时规则，以方法执行结束后为判断标准。

8.2　Cron 表达式

一个 Cron 表达式由 6 至 7 个有空格分隔的时间元素组成：{ 秒 } { 分 } { 时 } { 日 } { 月 } { 周 } { 年 (可选)}。

例如：0 30 01 * * ?

表示为：每天的凌晨 1 点 30 执行指定的任务。

每位时间元素描述

位	字段	是否必须	取值范围	允许的特殊字符
1	秒	是	0-59	, - * /
2	分	是	0-59	, - * /
3	时	是	0-23	, - * /
4	日	是	1-31	, - * / ?
5	月	是	1-12，或 JAN-DEC	, - * /
6	周	是	1-7，或 MON-SUN	, - * / ?
7	年	否	空，或 1970-2099	, - * /

特殊字符描述

*

对应字段上的所有值，例如在分钟字段，表示每一分钟。相应的有每一秒、每个月、每一年等。

?

表示该字段不设置。在两个字段只能选其一的情况下使用，例如日期和星期只能出现一个，那个不使用的那个字段就设置为 ?。

-

用于设置范围，上下界都包含，例如月份字段 8-10，表示 8，9，10 三个月份。

,

表示列表值（多值），如 MON、WED、FRI 表示周一、周三、周五，当取值不连续无法使用范围时，使用该字段。

/

配置步长 / 增量。格式为：开始 / 步长。 在秒位置上配置为 0/15，表示从 0 开始，15 为单位增加，在取值范围内的所有值，即 0、15、30、45，而 5/15，则取值为 5、20、35、50。

示例

- 0 * * * * *：每分钟。
- 0 0 * * * *：每小时。

- `*/10 * * * * *`：每 10 秒。
- `0 5/15 * * * *`：每小时的 5 分、20 分、35 分、50 分。
- `0 0 9,13 * * *`：每天的 9 点和 13 点。
- `0 0 8-10 * * *`：每天的 8 点、9 点、10 点。
- `0 0/30 8-10 * * *`：每天的 8 点、8 点半、9 点、9 点半、10 点。
- `0 0 9-17 * * MON-FRI`：每周一到周五的 9 点、10 点直到 17 点。
- `0 10,44 14 ? 3 WED`：每年 3 月份每周三的 14:10 和 14:44。
- `0 0 0 25 12 ?`：每年 12 月 25 日的 0 点 0 分 0 秒。
- `0 30 10 * * ? 2017`：2017 年每天的 10 点半。

8.3 QuartZ 分布式定时任务

在分布式模式下 Spring Task 并不适用，而 QuartZ 框架则提供了非常好的分布式定时任务解决方案。

在分布式模式中为了提高系统的可用性，往往一个应用会部署成多个实例，而每个实例中都拥有相同的定时任务，因此肯定会发生定时任务被重复执行的情况。为了解决这一问题，可以在定时任务处理具体业务逻辑之前先向 Redis 获得一个全局锁，各个实例在触发定时任务后谁先获得全局锁则执行业务逻辑，而其他实例则需要等待解锁，以此达到单实例执行定时任务的目的。但是定时任务是由时间驱动触发的，在获取全局锁时又会遇见并发请求的问题，在此可以使用消息队列将并发获取锁的过程转为队列获取。同时在各个定时任务实例中，如果有的实例在获取到全局锁后崩溃，则可能导致原本要执行的任务没有执行。现在需要在 Redis 中提供定时任务执行结果字段，实现崩溃转移机制，当某一实例在获得全局锁后发生崩溃，则将该任务交由其他实例执行。

QuartZ 并未与 `Spring Boot` 集成，在此以 `Spring 4.x` 与 `QuartZ 2.x.x` 版本作为示例配置。

Spring Boot 1.5.6.RELEASE 版本中集成了 Spring 4.3.10.RELEASE 版本的内核。

① 导入 MySQL 数据表。

分布式模式下 QuartZ 需要数据库存储定时任务信息，以达到集群调度的目的。目前支持：MySQL、H2、DB2、SQL Server、Oracle、Sybase 等数据库，在此以 MySQL 为例。

解压下载好的 QuartZ 压缩包，导入 `quartz-2.2.3\docs\dbTables` 目录下的 `tables_mysql_innodb.sql` 文件到 MySQL。

```
    // 在MySQL命令行模式下创建表
    mysql-> CREATE DATABASE quartz
    // 使用MySQL命令导入SQL文件
    mysql -h localhost -u root -p quartz < path\quartz-2.2.3\docs\dbTables\tables_mysql_
innodb.sql
```

导入成功后将获得如下表：

表名	介绍
QRTZ_CALENDARS	存储 Quartz 的日历信息
QRTZ_CRON_TRIGGERS	存储 Cron 表达式与时区信息
QRTZ_FIRED_TRIGGERS	存储已出发的触发器信息及任务执行信息
QRTZ_PAUSED_TRIGGER_GRPS	存储已暂停的触发器组信息
QRTZ_SCHEDULER_STATE	调度器状态
QRTZ_LOCKS	存储程序的悲观锁的信息
QRTZ_JOB_DETAILS	存储每一个已配置的任务详细信息
QRTZ_SIMPLE_TRIGGERS	存储触发器的重复次数、间隔，以及已触的次数
QRTZ_BLOG_TRIGGERS	触发器作为 Blob 类型存储
QRTZ_TRIGGERS	存储已配置的触发器信息
QRTZ_TRIGGERS_LISTENERS	触发器监听器

② 在 pom.xml 文件中引入依赖。

```
    <dependency>
      <groupId>org.quartz-scheduler</groupId>
      <artifactId>quartz</artifactId>
      <version>2.3.0</version>
    </dependency>

    <dependency>
      <groupId>org.quartz-scheduler</groupId>
      <artifactId>quartz-jobs</artifactId>
      <version>2.3.0</version>
    </dependency>

    <dependency>
       <groupId>org.springframework</groupId>
       <artifactId>spring-context-support</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
```

③ 新建 quartz.properties 文件并配置集群。

```
## 实例名称
org.quartz.scheduler.instanceName=spring-boot-quartz-demo
## 为每个实例分配一个不重复的id
org.quartz.scheduler.instanceId=AUTO
## 跳过更新检查
org.quartz.scheduler.skipUpdateCheck=true
## 线程池管理策略
org.quartz.threadPool.class=org.quartz.simpl.SimpleThreadPool
## 最大线程池数
org.quartz.threadPool.threadCount=5
## 线程优先级
org.quartz.threadPool.threadPriority=5
## 触发规则超时时间
org.quartz.jobStore.misfireThreshold=60000

## 是否开启集群模式
org.quartz.jobStore.isClustered=true
## 实例检查频率
org.quartz.jobStore.clusterCheckinInterval = 15000
```

④ 新建配置类整合 Spring 与 QuartZ 环境。

```
    public class AutoWiringSpringBeanJobFactory extends SpringBeanJobFactory implements
ApplicationContextAware {

        private transient AutowireCapableBeanFactory beanFactory;

        public void setApplicationContext(final ApplicationContext context) {
            beanFactory = context.getAutowireCapableBeanFactory();
        }

        @Override
        protected Object createJobInstance(final TriggerFiredBundle bundle) throws Exception {
            final Object job = super.createJobInstance(bundle);
            beanFactory.autowireBean(job);
```

```
        return job;
    }
}
```

AutowireCapableBeanFactory

定义了将容器中的 Bean 按某种规则（如按名字，类型的匹配等）进行自动装配的方法。

setApplicationContext

来自 Spring 的 ApplicationContextAware 接口，通过 getAutowireCapableBeanFactory() 获得 Bean 的依赖注入对象。

createJobInstance

来自 QuartZ 的 SpringBeanJobFactory 类，在创建任务实例时交由 Spring 自动注入。

⑤ 新建定时任务类。

```
public class MyTask implements Job {

    @Autowired
    private DemoService service;

    @Override
    public void execute(JobExecutionContext context) throws JobExecutionException {
        service.server("1");
    }

}
```

实现 Job 接口后在 exclude() 方法中执行定时任务逻辑。

DemoService 是一个普通的 Spring 管理类，通过注解 @Autowired 注入，如果没有 AutoWiringSpring BeanJobFactory 整合的 Spring 与 QuartZ，则注入失败，无法获取 DemoService 实例。

为了测试，DemoService 的 server() 方法只打印了当前时间。

⑥ 新建 quartz.xml 文件，配置定时任务。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"
```

```
    default-lazy-init="false">
    <!-- 数据源 -->
    <bean id="quartzDataSource" class="org.springframework.jdbc.datasource.SimpleDriver DataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/quartz?useSSL= false" />
        <property name="username" value="root" />
        <property name="password" value="pwd" />
    </bean>
    <!-- 触发器 -->
    <bean id="trigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
        <property name="jobDetail">
            <bean class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
                <!-- 开启定时任务持久化 -->
                <property name="Durability" value="true" />
                <!-- 定时任务地址 -->
                <property name="jobClass" value="org.book.task.MyTask" />
            </bean>
        </property>
        <!-- Cron表达式 -->
        <property name="cronExpression">
            <value>0/10 * * * * ?</value>
        </property>
    </bean>
    <!-- 调度工厂 -->
    <bean id="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <!-- 使触发器生效 -->
        <property name="triggers">
            <list>
                <ref bean="trigger" />
            </list>
        </property>
        <!-- 指定Quartz配置文件地址 -->
        <property name="configLocation" value="classpath:quartz.properties" />
        <!-- 启动时延期3秒开始任务 -->
        <property name="startupDelay" value="3" />
        <!-- 使数据源生效 -->
        <property name="dataSource" ref="quartzDataSource" />
        <!-- 使数自动注入配置生效 -->
        <property name="jobFactory">
            <bean class="org.book.config.AutoWiringSpringBeanJobFactory" />
        </property>
    </bean>
</beans>
```

⑦ 在 Spring boot 入口处指定 QuartZ 配置文件。

```
@SpringBootApplication
@ImportResource("quartz.xml")
public class SpringBootTaskApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootTaskApplication.class, args);
    }
}
```

⑧ 测试运行。

启动多个该 Spring Boot 定时任务应用后，观察控制台输出内容，会发现在同一时间内只有一个实例在执行定时任务一输出当前时间，而其他的则在等待。

将当前正在执行定时任务的实例强行关闭后，会发现其他正在运行的某一实例接管了定时任务，继续打印当前时间。

本章示例代码详见异步社区网站本书页面。

第9章　分布式会话

由于 HTTP 是无状态协议，每次发起请求时服务端并不知道各个请求之间的关系，为解决这个问题，引入了 Session 与 Cookie 配合记录客户端（浏览器）所发起的请求。

当打开浏览器发起 HTTP 请求时，服务端的 Session 生成一个全局统一标识（session_id），并将这个标识发送给客户端存储于 Cookie 中。基于该统一标识便可管理当前浏览器所发起的请求之间的关系。Spring Boot 中的基于内嵌的 Tomcat 容器将 Session 存储于内存中，在分布式环境中由于各微服务应用运行于不同的环境，所以各 Session 之间无法互通。

Spring Session 通过标准的 Servlet Filter（过滤器）拦截所有的 Web 请求，并且重写了 HttpServletRequest 的 getSession() 方法，将 Session 交由 Redis 存储，多个微服务应用使用同一 Redis 管理的 Session，从而最终实现统一的分布式会话管理。

由于 Dubbo 的各微服务模块并非在 Web 容器中运行，所以分布式 Session 会话在 Spring Cloud 中使用，不过在设计架构时尽量还是使用无状态的服务接口。

① 在 pom.xml 中引入依赖。

```
<dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-data-redis</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

② 在 appaction.properties 中配置。

```
spring.redis.database=0
spring.redis.host=localhost
spring.redis.port=6379
```

```
spring.session.store-type=redis
```

spring.session.store-type

指定 Session 的存储服务提供方。

③ 编写 Controller 测试。

```
@RequestMapping("put")
public Map<String, Object> putSession(HttpSession session, @RequestParam("key") String key,
    @RequestParam("value") String value) {
  Map<String, Object> map = new HashMap<>();
  session.setAttribute(key, value);
  map.put("id", session.getId());
  map.put("session", value);
  return map;
}

@RequestMapping("get")
public Map<String, Object> getSession(HttpSession session, @RequestParam ("key")
String key) {
  Map<String, Object> map = new HashMap<>();
  map.put("id", session.getId());
  map.put("session", session.getAttribute(key).toString());
  return map;
}
```

调用 putSession 请求后，查询 Redis 便可发现，Spring Session 已经将刚才存入 Session 的内容写入了 Redis 中。现在只需在其他微服务应用中进行相同的配置便可从共享 Redis 中获取 Session 信息。

```
127.0.0.1:6379> keys *
1) "spring:session:sessions:0515ff04-2f1a-4fc2-87ae-b83c3e40ad5e"
2) "spring:session:sessions:expires:0515ff04-2f1a-4fc2-87ae-b83c3e40ad5e"
3) "spring:session:expirations:1504905480000"
```

第10章　消息队列

10

10.1　安装及配置 RabbitMQ

10.2　配置及使用

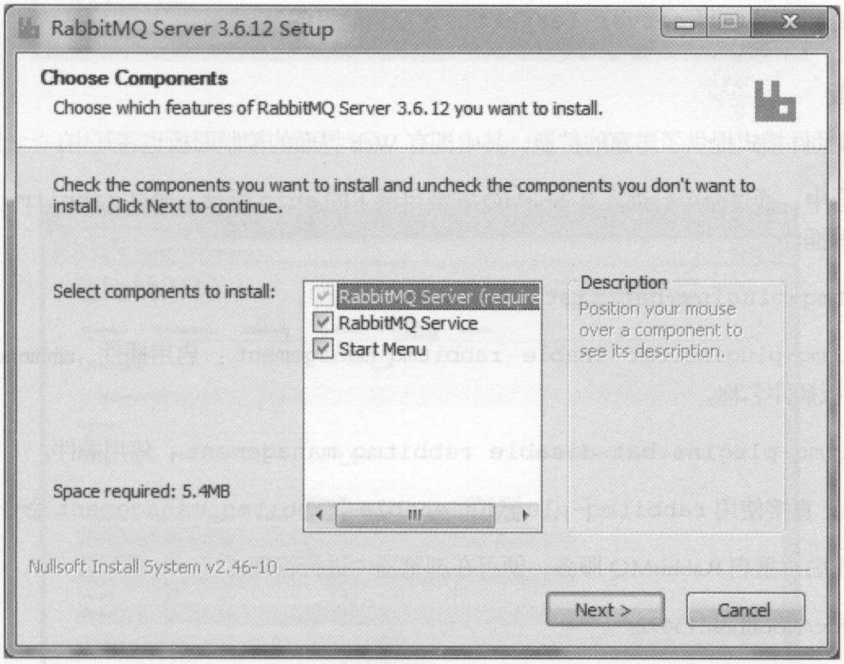
Spring Boot 基于 AMQP 协议封装了 RabbitMQ 消息中间件，提供了开箱即用的消息服务。

AMQP：Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在。并且 AMQP 拥有面向消息、队列、路由（包括点对点和发布 / 订阅）、可靠性、安全等特征。

RabbitMQ：由 Erlang 语言编写，是实现了 AMQP 协议的消息中间件。最初起源于金融系统，用于在分布式系统中存储转发消息，并且拥有较好的易用性、扩展性、高可用性。

10.1 安装及配置RabbitMQ

Windows平台



直接通过官网提供的 Windows 安装包进行安装，RabbitMQ 已经预制好了常用脚本以便于管理，安装成功后可以在 Windows 的开始菜单搜索获得。

- `RabbitMQ Server - start`：启动服务。

- `RabbitMQ Server - stop`：关闭服务。
- `RabbitMQ Server - remove`：删除服务。

Linux平台

```
## erlang 语言运行环境
yum install erlang
## 安装下载好的RabbitMQ
yum install rabbitmq-server-xxxxx.rpm
```

操作命令如下。

- `service rabbitmq-server start`：启动服务。
- `service rabbitmq-server stop`：关闭服务。
- `service rabbitmq-server restart`：启重启动服务。

开启控制面板

RabbitMQ 以插件模式提供了丰富的扩展，其中拥有 Web 界面的控制面板也在其中。

Windows 平台中，通过命令行进入 RabbitMQ 安装目录下的 `sbin` 目录，通过 `rabbitmq-plugins.bat` 脚本管理插件。

- `.\rabbitmq-plugins.bat list`：列出目前可用插件。
- `.\rabbitmq-plugins.bat enable rabbitmq_management`：启用插件，rabbitmq_management 是控制面板插件名称。
- `.\rabbitmq-plugins.bat disable rabbitmq_management`：停用插件。

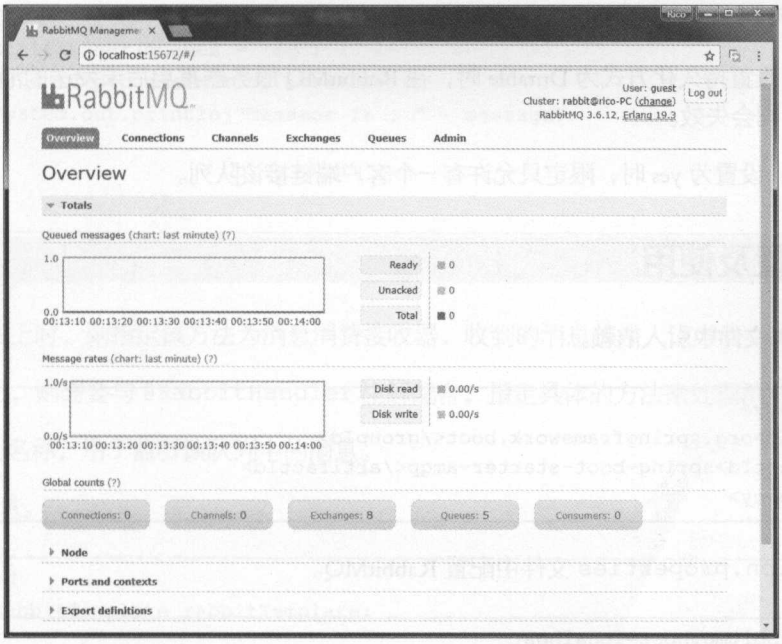
Linux 平台中，直接使用 `rabbitmq-plugins enable rabbitmq_management` 命令安装插件。

插件安装完成后，重启 RabbitMQ 服务，便可在浏览器中进行管理。

访问地址：http://localhost:15672/

默认用户：guest

默认密码：guest



配置队列

RabbitMQ 只允许在已经注册的队列中发送消息，在 RabbitMQ 控制面板的 Queues 目录下添加队列。

- Name：队列名称。
- Durability：设置持久化方式为 Durable 时，在 RabbitMQ 服务器重启后未发送的消息将会被保存，而 Transient 则会失效。
- Auto delete：设置为 yes 时，限定只允许有一个客户端链接该队列。

10.2　配置及使用

① 在 pom.xml 文件中引入依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

② 在 appaction.properties 文件中配置 RabbitMQ。

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
```

spring.rabbitmq.host

链接地址。

spring.rabbitmq.port

访问端口。

③ 在 Spring Boot 入口处开启 RabbitMQ 支持。

使用注解 @EnableRabbit 开启。

```
@SpringBootApplication
@EnableRabbit
public class SpringBootMqApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootMqApplication.class, args);
    }
}
```

④ 编写服务消费者。

```
@Component
public class MQReceiver {
```

```
    @RabbitListener(queues = "mq")
    public void process(String message) {
        System.out.println("Message is : " + message);
    }

}
```

@RabbitListener

该注解用在方法上时，则指定该方法为消息消费接收器，收到的消息将注入该方法中的形参中。

如果用在类上时，则需要与 @RabbitHandler 注解配合，指定具体的方法来处理消费消息。

queues：队列名称，用于监听该队列中的消息。

⑤ 测试发送消息。

```
@Autowired
private RabbitTemplate rabbitTemplate;

public void sender() {
    rabbitTemplate.convertAndSend("mq", "message content");
}
```

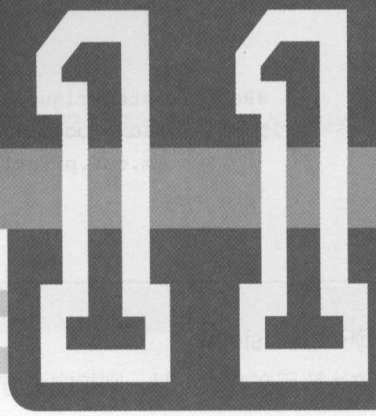
使用 RabbitTemplate 的 convertAndSend() 方法发送消息。

- 第一个参数为队列名称，需要与消费者 RabbitListener 注解中的 queues 参数一致。
- 第二个参数则为消息的具体内容，可以是任意类型数据，同样需要确保与消费者接受消息方法中的参数类型一致。

当有多个消费者或消息提供者实例同时存在时，发送的消息经过 rabbit 路由处理，最终消息将均匀地分布在各个消费终端中，这一点与 Dubbo、Spring Cloud 的负载均衡机制类似，同样也可以基于消息队列的特性构建异步通信的分布式架构。

本章示例代码详见异步社区网站本书页面。

第11章　构建Web应用



使用 Spring Boot 可以非常轻松地构建 Web 应用，也可轻松管理静态资源文件及页面模板。

新建 Spring Boot 应用时勾选 Spring Boot MVC 模块，便会发现在 src/main/resources 目录下多出 static 与 templates 两个文件夹，前者用于存放静态文件，例如在 static 文件中放入名为 pic.jpg 的图片，则对应的访问地址为 http://localhost:8080/pic.jpg；而 templates 中则用于存放模板文件，如 *.jsp 等。

Spring Boot 提供了默认配置的模板引擎主要有以下几种：

- Thymeleaf
- FreeMarker
- Velocity
- Groovy
- Mustache

Spring Boot 官方并不推荐使用 JSP 模板引擎，所以这里以 FreeMarker 为例进行讲解。

① 在 pom.xml 中添加 FreeMarker 依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

② 在 application.properties 文件中配置。

```
spring.freemarker.cache=false
spring.freemarker.charset=UTF-8
spring.freemarker.content-type=text/html
spring.freemarker.expose-request-attributes=true
spring.freemarker.expose-session-attributes=true
spring.freemarker.request-context-attribute=request
spring.freemarker.suffix=.ftl
spring.freemarker.template-loader-path=classpath:/templates/
```

spring.freemarker.cache
是否开启模板缓存。

spring.freemarker.charset
指定目标使用的编码。

spring.freemarker.expose-request-attributes
设定所有 request 的属性在合并到模板的时候，是否要都添加到 model 中。

spring.freemarker.expose-session-attributes
设定所有 HttpSession 的属性在合并到模板的时候，是否要都添加到 model 中。

spring.freemarker.request-context-attribute
指定 RequestContext 属性的名。

spring.freemarker.suffix
指定模板文件的后缀。

spring.freemarker.template-loader-path
指定模板文件存放的位置。

spring.freemarker.content-type
设定 Content-Type 的类型。

③ 编写模板文件。

新建 hello.ftl 模板文件并存放于 resources/templates 目录下。

```
<!DOCTYPE html>
```

```
<html>
<body>
    获取内容: ${message}
<br>
    调用方法: ${time?datetime}
<br>
    字符串: ${"输出字符串"}
<br>
    计算: ${1 + 2}
<br>
    判断: <#if 1 == 2 > 等于 <#else> 不等于 </#if>
<br>
    遍历: <#list [1,2,3,4] as item> ${item} <#sep>,  </#list>
</body>
</html>
```

④ 编写 Controller。

```
@RequestMapping(value = "/")
public ModelAndView hello() {
  Map<String, Object> map = new HashMap<>();
  map.put("message", "你好 FreeMarker");
  map.put("time", new Date());
  return new ModelAndView("hello", map);
}
```

返回的 ModelAndView 包含了两个参数，第一个参数指定了模板具体名称，第二个参数为在 request 作用域转发时合并的数据。

本章示例代码详见异步社区网站本书页面。

第12章　异常处理

在启动应用时会发现在控制台打印的日志中出现了两个路径为{[/error]}的访问地址，当系统中发生异常错误时，Spring Boot会根据请求方式分别跳转到以JSON格式或以界面显示的/error地址中显示错误信息。

```
    --- [   restartedMain] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/
error]}" ....
    --- [   restartedMain] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/
error]" ....
```

使用AJAX方式请求时返回的JSON格式错误信息。

```
{
    "timestamp": 1505942837186,
    "status": 500,
    "error": "Internal Server Error",
    "exception": "java.lang.RuntimeException",
    "message": "No message available",
    "path": "/err"
}
```

使用浏览器请求时返回的错误信息界面。

```
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a
fallback.

Thu Sep 21 05:23:04 CST 2017
There was an unexpected error (type=Internal Server Error, status=500).
No message available
```

自定义异常

一个完整的系统在对外提供接口服务时，错误信息及错误码必不可少，Spring Boot 对错误的封装并不能够满足需求，所以需要自行定义。

① 在 POM.xml 文件中引入 JSON 序列化依赖。

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.38</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
```

② 新建错误信息实体。

```
public class ExceptionEntity implements Serializable {

    private static final long serialVersionUID = 1L;

    private String message;

    private int code;

    private String error;

    @JSONField(format = "yyyy-MM-dd hh:mm:ss")
    private Date timestamp = new Date();

    private String path;

  //getter and setter
}
```

③ 新建自定义异常。

```
public class BasicException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    private int code = 0;
```

```
    public BasicException(int code, String message) {
        super(message);
        this.code = code;
    }

    public int getCode() {
        return this.code;
    }

}

public class BusinessException extends BasicException {

    private static final long serialVersionUID = 1L;

    public BusinessException(int code, String message) {
        super(code, message);
    }

}
```

BasicException 继承了 RuntimeException，并在原有的 Message 基础上增加了错误码 code 的内容。而 BusinessException 则是在业务中具体使用的自定义异常类，起到了对不同的异常信息进行分类的作用。

④ 新建 error.ftl 模板文件用于显示错误信息。

```
<!DOCTYPE html>
<html>
<body>
    错误信息: ${exception.message}
<br>
    错误码: ${exception.code}
<br>
    时间: ${exception.timestamp?datetime}
<br>
    请求路径: ${exception.path}
<br>
    异常: ${exception.error}
</body>
</html>
```

⑤ 编写全局异常控制类。

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(value = BasicException.class)
    public ModelAndView errorHandler(HttpServletRequest request, BasicException exception,
                                     HttpServletResponse response) {
        ExceptionEntity entity = new ExceptionEntity();
        entity.setMessage(exception.getMessage());
        entity.setPath(request.getRequestURI());
        entity.setCode(exception.getCode());
        entity.setMessage(exception.getMessage());
        entity.setPath(request.getRequestURI());
        entity.setError(exception.getClass().getSimpleName());
        if (!(request.getHeader("accept").contains("application/json")
                || (request.getHeader("X-Requested-With") != null
                && request.getHeader("X-Requested-With").contains("XMLHttpRequest")))) {
            ModelAndView modelAndView = new ModelAndView("error");
            modelAndView.addObject("exception", entity);
            return modelAndView;
        } else {
            try {
                response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
                response.setCharacterEncoding("UTF-8");
                response.setHeader("Content-Type", "application/json");
                response.getWriter().write(JSON.toJSONString(entity));
            } catch (IOException e) {
                e.printStackTrace();
            }
            return null;
        }
    }

    @ResponseBody
    @ExceptionHandler(value = BindException.class)
    public ExceptionEntity validExceptionHandler(BindException exception, HttpServletRequest
request, HttpServletResponse response){
        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();
        Map<String,String> errors = new HashMap<>();
        for (FieldError error:fieldErrors){
            errors.put(error.getField(),error.getDefaultMessage());
        }
        ExceptionEntity entity = new ExceptionEntity();
        entity.setMessage(JSON.toJSONString(errors));
```

```
        entity.setPath(request.getRequestURI());
        entity.setCode(500);
        entity.setError(exception.getClass().getSimpleName());
        response.setStatus(500);
        return entity;
    }
```

@ControllerAdvice

作用于类上，用于标识该类用于处理全局异常。

@ExceptionHandler

作用于方法上，用于对拦截的异常类型进行处理。value 属性用于指定具体的拦截异常类型，如果有多个 ExceptionHandler 存在，则需要指定不同的 value 类型，由于异常类拥有继承关系，所以 ExceptionHandler 会首先执行在继承树中靠前的异常类型。

BasicException 异常继承自 RuntimeException，如果两者同时存在，则只执行拦截了 RuntimeException 的处理器

该方法捕获到了系统中的异常，并将信息存入 ExceptionEntity 中，通过 HTTP 头信息判断是 AJAX 请求还是普通请求，最终根据请求类型以不同的方式输出异常结果。

BindException

该异常来自于第 7 章所介绍的表单验证框架 Hibernate validation，当字段验证未通过时会抛出此异常，通过全局捕获该异常可以统一处理来自表单验证的错误信息，这样在 Controller 中就不必要再添加 BindingResult。BindException 适用于 url 传参时进行参数验证。但如果使用 body 进行传参时，则需要拦截 NestedRuntimeException 异常。

⑥ 编写测试 Controller。

```
@RequestMapping(value = "err")
public void error() {
  throw new BusinessException(400, "错误信息");
}
```

启动应用后，使用 postman 软件并在 headers 中添加 accept:application/json 头信息模拟 AJAX 请求。

返回结果如下所示：

```
{
    "code": 400,
    "error": "BusinessException",
    "message": "错误信息",
```

```
    "path": "/err",
    "timestamp": "2017-09-21 08:35:51"
}
```

直接在浏览器中访问，返回信息如下：

```
错误信息：错误信息
错误码：400
时间：2017-9-21 8:32:42
请求路径：/err
异常：BusinessException
```

本章示例代码详见异步社区网站本书页面。

第13章　安全认证

13.1　OAuth2.0 协议介绍

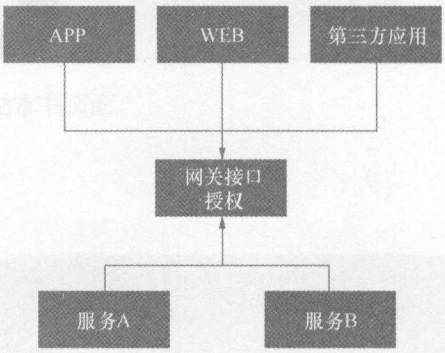
13.2　授权模式

13.3　在 Dubbo 中使用 OAuth 2.0

13.4　在 Spring Cloud 中使用 OAuth 2.0

在项目中各个服务模块产生的接口并不希望被第三方滥用，所以在向外暴露接口的网关服务中增加了拦截器，并对其进行了简单的安全限制。

但在实际需求中，微服务所产生的接口可能会服务于本系统中的多个客户端，如：iOS、Android、Web、PC 等，或者将这些接口开放给第三方系统使用。在提供服务的同时需要一套授权机制来保护安全，在网关服务中增加拦截器的做法显然有一些简单，为此 Spring 提供了 Spring Security 框架来解决这一问题，并且实现了 OAuth2.0 协议对各类客户端进行授权管理。



13.1　OAuth2.0 协议介绍

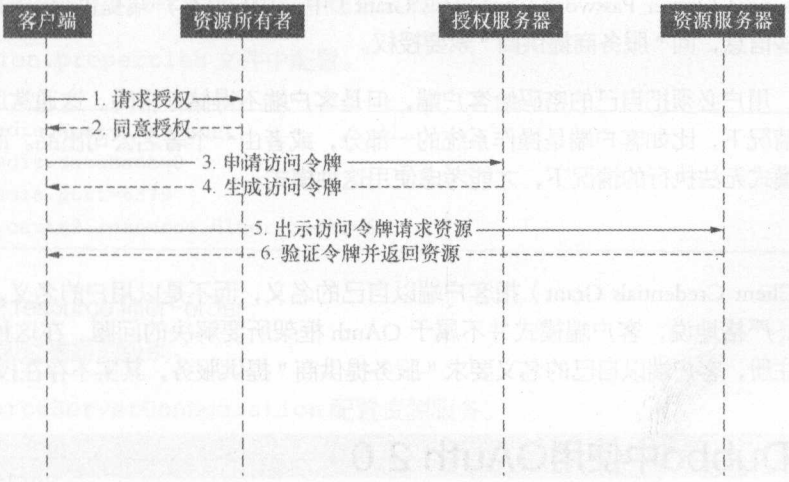
OAuth：OAuth（开放授权）是一个开放标准，允许用户授权第三方移动应用访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方移动应用或分享他们数据的所有内容。

OAuth 2.0 是 OAuth 协议的下一版本，但不向后兼容 OAuth 1.0。OAuth 2.0 关注客户端开发者的简易性，同时为 Web 应用、桌面应用和手机等提供专门的认证流程，目前主要使用的版本是 2.0 版。

协议参与者

- Resource Owner：资源所有者，对资源具有授权能力的人。
- Resource Server：资源服务器，它存储资源，并处理对资源的访问请求。
- Client：第三方应用，它获得 RO 的授权后便可以去访问 RO 的资源。
- Authorization Server：授权服务器。

基本授权流程



① 客户端向资源所有者（用户）请求授权。

② 客户端收到授权许可。

③ 客户端与授权服务器进行身份认证并出示授权许可请求访问令牌。

④ 授权服务器验证客户端身份并验证授权许可，若有效则颁发访问令牌。

⑤ 客户端从资源服务器请求受保护资源并出示访问令牌进行身份验证。

⑥ 资源服务器验证访问令牌，若有效则满足该请求。

13.2 授权模式

OAuth2.0 定义了四种授权模式。

授权码模式

授权码模式（authorization code）是功能最完整、流程最严密的授权模式。它的特点就是通过客户端的后台服务器，与 " 服务提供商 " 的认证服务器进行互动。

简化模式

简化模式（implicit grant type）不通过第三方应用程序的服务器，而是直接在浏览器中向认证服务器申请令牌，跳过了 " 授权码 " 这个步骤，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且客户端不需要认证。

密码模式

密码模式（Resource Owner Password Credentials Grant）中，用户向客户端提供自己的用户名和密码。客户端使用这些信息，向 " 服务商提供商 " 索要授权。

在这种模式中，用户必须把自己的密码给客户端，但是客户端不得储存密码。这通常用在用户对客户端高度信任的情况下，比如客户端是操作系统的一部分，或者由一个著名公司出品。而认证服务器只有在其他授权模式无法执行的情况下，才能考虑使用这种模式。

客户端模式

客户端模式（Client Credentials Grant）指客户端以自己的名义，而不是以用户的名义，向 " 服务提供商 " 进行认证。严格地说，客户端模式并不属于 OAuth 框架所要解决的问题。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求 " 服务提供商 " 提供服务，其实不存在授权问题。

13.3　在Dubbo中使用OAuth 2.0

Dubbo 框架中的网关模块代理了 Dubbo 服务的 Web 调用，并将这些服务转化为 HTTP 协议的 API 接口，所以本章使用 Spring Boot 进行 OAuth2.0 配置。

① 在 pom.xml 文件中引入依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

spring-boot-starter-security

spring 安全框架。

spring-security-oauth2

spring 安全框架的 OAuth2.0 协议扩展。

spring-boot-starter-data-redis

引入 Redis 缓存高频调用的 Access Token 信息以提高性能。

② 在 appaction.properties 文件中配置。

```
spring.redis.host=localhost
spring.redis.database=0
spring.redis.port=6379
security.oauth2.resource.filter-order=6
```

security.oauth2.resource.filter-order

设置 OAuth 框架过滤器的级别。

③ 新建 ResourceServerConfiguration 配置资源服务。

```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
http
.authorizeRequests()
.antMatchers(HttpMethod.OPTIONS).permitAll() .antMatchers("/public/{id}").permitAll()
.and()
.authorizeRequests()
.anyRequest()
.authenticated()
.and()
.httpBasic();
    }
}
```

每个提供服务的 URL 接口便是资源，通过继承 ResourceServerConfigurerAdapter 类并重写 configure 方法完成对资源的配置，有点类似于过滤器限制所要链接的 URL。并通过 @EnableResourceServer 注解表示该应用为资源服务。

HttpSecurity 是 SecurityBuilder 接口的实现类，用于构建 HTTP 安全相关的配置，并且可以通过链式编程的方式完成自定义的配置。

- authorizeRequests：注册 url 规则权限匹配
- antMatchers：匹配请求
- permitAll：不进行权限拦截

- anyRequest：任何请求
- authenticated：进行认证
- httpBasic：开启 HTTP 基础认证
- and：连接符

④ 新建 AuthorizationServerConfiguration 配置授权服务。

```
@Configuration
@EnableAuthorizationServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private RedisConnectionFactory redisConnectionFactory;

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.withClientDetails(new ClientDetailsService() {
            @Override
            public ClientDetails loadClientByClientId(String clientId) throws Client Registration
Exception {
                BaseClientDetails client = new BaseClientDetails();
                if (clientId.equals("web")) {
                    client.setAuthorizedGrantTypes(Arrays.asList("client_credentials",
"refresh_token"));
                } else if (clientId.equals("app")) {
                    client.setAuthorizedGrantTypes(Arrays.asList("password", "refresh_
token"));
                } else {
                    throw new NoSuchClientException("客户端id错误");
                }
                client.setClientId(clientId);
                client.setClientSecret("123456");
                client.setScope(Arrays.asList("select"));
                client.setAccessTokenValiditySeconds(24 * 60 * 60);
                client.setRefreshTokenValiditySeconds(48 * 60 * 60);
                client.setAuthorities(AuthorityUtils.commaSeparatedStringToAuthority
List("ROLE_CLIENT"));
                return client;
            }
```

```
        });
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints
        .tokenStore(tokenStore())
        .authenticationManager(authenticationManager);
    }

    @Bean
    public TokenStore tokenStore(){
        return new RedisTokenStore(redisConnectionFactory);
    }

}
```

通过继承 AuthorizationServerConfigurerAdapter 配置授权服务策略，并使用 @EnableAuthorizationServer 注解标识该应用是授权服务应用。

ClientDetailsServiceConfigurer

配置客户端授权管理信息，如果拥有多个客户端（开放平台系统），可以根据提供的 clientId 从数据库中查询，并将各相关配置提交给 BaseClientDetails。

- setClientId：客户端 id，必要
- setClientSecret：客户端密码
- setScope：权限范围
- setAccessTokenValiditySeconds：token 创建时的超时时间
- setRefreshTokenValiditySeconds：token 刷新时间
- setAuthorities：客户端拥有的身份
- setAuthorizedGrantTypes：客户端拥有的授权方式
  - password：密码模式
  - client_credentials：客户端模式
  - refresh_token：刷新 token
  - implicit：简化模式

AuthorizationServerEndpointsConfigurer

用于配置授权管理器策略与存储 token 的策略，通过注入 TokenStore 指定将 token 信息存储在 redis 中。

@EnableGlobalMethodSecurity

设置全局方法安全控制，通过prePostEnabled = true设置在方法调用之前拦截，并进行权限验证。

⑤ 新建 SecurityConfiguration 进行安全配置。

```
    @Configuration
    @EnableWebSecurity
    public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

        @Bean
        @Override
        protected UserDetailsService userDetailsService() {
            return new UserDetailsService() {
                @Override
                public UserDetails loadUserByUsername(String username) throws UsernameNot
FoundException {
                    if (!username.equals("book")) {
                        throw new UsernameNotFoundException("用户错误");
                    }
                    String password = new BCryptPasswordEncoder().encode("123456");
                    return new User(username, password, AuthorityUtils.createAuthorityList
("ROLE_USER"));
                }
            };
        }

        @Bean
        public FilterRegistrationBean corsFilter() {
            UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
            CorsConfiguration config = new CorsConfiguration();
            config.setAllowCredentials(true);
            config.setAllowedOrigins(Arrays.asList("http://localhost:8080"));
            config.addAllowedHeader("*");
            config.addAllowedMethod("*");
            source.registerCorsConfiguration("/**", config);
            FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));
            bean.setOrder(Ordered.HIGHEST_PRECEDENCE);
            return bean;
        }

        @Bean
        public PasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
```

```
    }
}
```

WebSecurityConfigurerAdapter 来自于 Spring Web 的安全管理框架，用于对访问 web 的用户身份进行安全配置。需要配合 @EnableWebSecurity 注解开启安全验证。

UserDetailsService

对应之前配置的名为 app 的客户端所使用的 password 授权模式，在每次提交用户名与密码申请 access_token 时，Spring Security 将与 UserDetailsService 所提供的用户信息做校验。

User 的接口实现类 UserDetails() 用于封装用户名、密码、角色等信息，并且需要按照格式 ROLE_* 指定用户角色。

这里是将用户的信息写死在代码中，在实际开发中可以从数据库中读取用户信息并且提交给 UserDetails。

FilterRegistrationBean

Spring Security 安全框架默认开启了 CORS（跨域资源共享）机制进行跨域访问控制。配合之前在 ResourceServerConfigurerAdapter 中对 OPTIONS 类型请求不进拦截，这里新建一个拦截器完成对 CORS 的配置。

浏览器在发起跨域请求时，会首先向目标服务器发起一个 OPTIONS 请求，用于询问当前请求的域是否允许，如果允许则返回数据，否则将提示跨域错误，并拦截返回信息。

PasswordEncoder

一般情况下数据库中存储的用户密码会经过加密处理，但是用户申请 access_token 提交的则是明文密码，如果不指定校验密码时的加密策略肯定无法通过验证。这里通过注入 PasswordEncoder 来告知 Spring Security 使用 BCrypt 策略进行加密。

同样，对应创建用户存储密码时可以使用 new BCryptPasswordEncoder().encode(password) 对密码进行加密处理。

⑥ 编写测试 Controller。

```
    @Autowired
    private RedisTemplate redisTemplate;

    @PreAuthorize("hasRole('ROLE_USER') or #oauth2.clientHasRole('ROLE_CLIENT')")
    @RequestMapping("/private/{id}")
    public String privateIndex(Principal principal, @PathVariable("id") int id) {
        TokenStore tokenStore = new RedisTokenStore(redisTemplate.getConnectionFactory());
        Collection<OAuth2AccessToken> oAuth2AccessTokens = tokenStore.findTokensByCl
```

```
ientIdAndUserName("app", principal.getName());
            oAuth2AccessTokens.forEach(oAuth2AccessToken -> {
                tokenStore.removeAccessToken(oAuth2AccessToken);
            });
            return "success:" + id;
        }

    @RequestMapping(value = "/public/{id}")
    public String publicIndex(@PathVariable String id) {
        return "success : " + id;
    }
```

@PreAuthorize

对应 @EnableGlobalMethodSecurity 注解，在该方法调用之前进行拦截，并通过指定的 Spring EL 表达式验证权限。该注解可以应用在任意 Spring 管理的方法之上。

- hasRole('ROLE_USER')：限定用户角色为 ROLE_USER 访问。
- or、and：连接符。
- #oauth2.clientHasRole('ROLE_CLIENT')"：限定客户端角色为 ROLE_CLIENT 访问。
- hasIpAddress(192.168.0/24)：限定 ip 地址为 192.168.0.24 的 IP 访问。
- hasAuthority('user')：限定名为 user 的用户访问。

Principal

在访问受保护的资源时，需要提交 access_token 完成身份验证，通过在形参中注入 Principal 便可获得当前访问的 access_token 所对应的用户名称。

TokenStore

当用户修改了用户名或密码后，会涉及到重新生成 access_token 的需求。之前在配置 Spring Security 的权限管理策略时通过注入 TokenStore 指定使用 redis 存储 token 信息，同样也可以使用 TokenStore 找到 token 并删除。

⑦ 授权测试。

部署应用后访问 http://localhost:8080/public/1 成功获得返回结果，但是当访问 /private/1 时拦截器生效，提示需要授权才可访问。

```
{
    "error": "unauthorized",
    "error_description": "Full authentication is required to access this resource"
}
```

之前设置过 web 和 app 两个客户端，可以通过 POST 方式获取，但 OAuth2 协议中客户端信息需要使用 Basic Auth 方式传递，为了方便测试推荐使用 Chrome 浏览器的 Postman 插件。

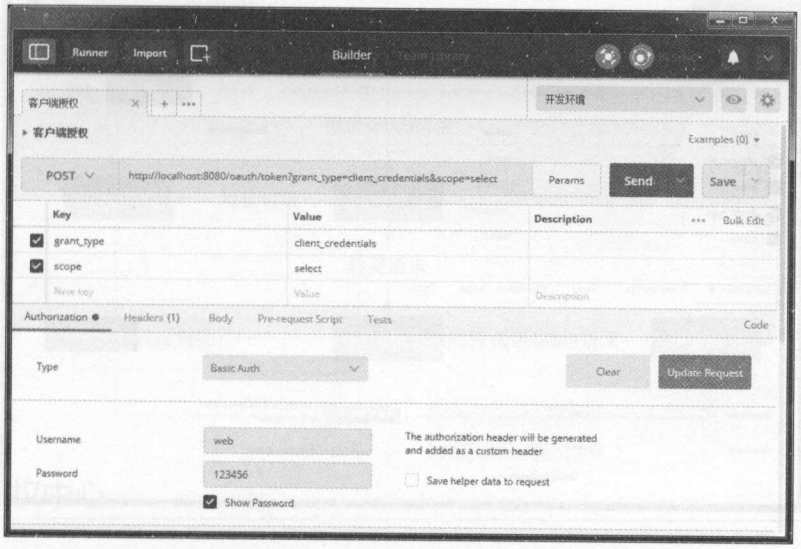
获取客户端模式授权

请求方式：POST

请求地址：http://localhost:8080/oauth/token

URL 参数：授权类型（grant_type）、权限范围（scope）

Basic Auth 参数：客户端用户名（clientId）、客户端密码（Secret）



返回结果如下：

```
{
    "access_token": "120bd424-0eec-471e-a858-4243ad9611cd",
    "token_type": "bearer",
    "expires_in": 65307,
    "scope": "select"
}
```

- access_token：授权后的 token
- token_type：token 类型
- expires_in：token 过期时间
- scope：权限范围

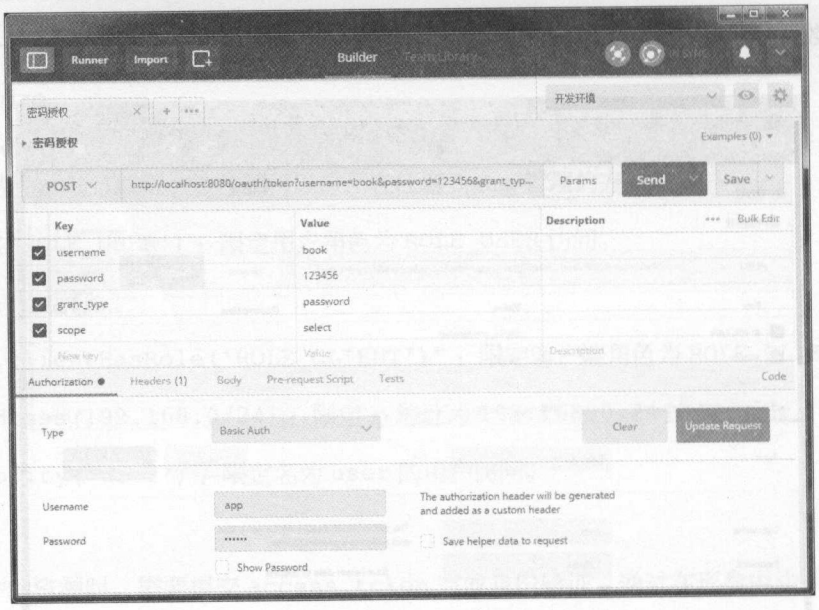
获取密码模式授权

请求方式：POST

请求地址：http://localhost:8080/oauth/token

URL 参数：授权类型（grant_type）、权限范围（select）、用户名（username）、用户密码（password）

Basic Auth 参数：客户端用户名（clientId）、客户端密码（Secret）



返回结果：

```
{
    "access_token": "95cd8715-e48c-4876-ba32-617d62032ac8",
    "token_type": "bearer",
    "refresh_token": "e4cc28ce-2fb2-4d96-9998-106e1e7dcda1",
    "expires_in": 65215,
    "scope": "select"
}
```

返回结果与客户端授权模式基本一致，只多了一个用于增加 token 过期时间的 `refresh_token` 值。

访问数据

http://localhost:8080/private/1?access_token=120bd424-0eec-471e-a858-4243ad9611cd

在访问受保护 URL（资源）时，只需在 URL 中传递获取到的 `access_token` 便可成功获取数据。

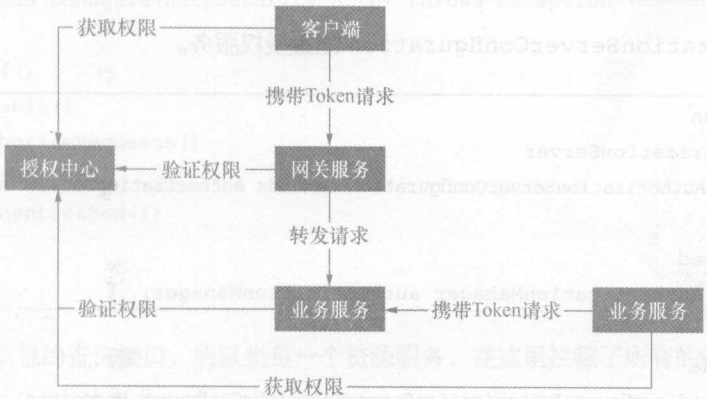
或将 access_token 加入名为 Bearer 的请求头中。

请求头示例：Authorization: Bearer fde0e94d-a20b-45b1-8f69-ae45aaec35ad

本章示例代码详见异步社区网站本书页面。

13.4 在Spring Cloud 中使用OAuth 2.0

Spring Cloud 框架中的每一个服务模块均是一个独立的 Web 应用，即是独立的资源提供方。而网关服务（Zuul）并不是资源的实际拥有者只是转发了请求，所以在各模块之间互相调用时，重点需要处理好授权信息的转发。



13.4.1 授权中心

授权中心负责生成及验证认证 Token，由于每一次的调用都需要验证权限，为了性能考虑所以将此服务独立出来。

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Server 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-auth
version: 0.0.1-SNAPSHOT
packaging: jar
```

pom依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-security</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-oauth2</artifactId>
    </dependency>
```

② 新建 AuthorizationServerConfiguration 配置授权服务。

```
    @Configuration
    @EnableAuthorizationServer
    public class AuthorizationServerConfiguration extends AuthorizationServerConfigurerAdapter{

        @Autowired
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints.authenticationManager(authenticationManager);
        }

        @Override
        public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
            clients.inMemory()
            .withClient("app")
            .scopes("app")
            .secret("123456")
            .authorizedGrantTypes("password", "authorization_code", "refresh_token")
              .and()
            .withClient("client")
            .scopes("client")
            .authorizedGrantTypes("implicit");
        }

    }
```

ClientDetailsServiceConfigurer

使用 inMemory() 以内存存储的方式配置客户端信息。

这里配置了名为 app 和 client 两个客户端，其中 client 使用简单授权模式，用于各微服务模块之间进行调用时的权限认证。而 app 则是提供给外部应用进行授权使用。

③ 新建 ResourceServerConfiguration 配置资源服务。

```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
        .csrf()
        .disable()
        .authorizeRequests()
        .anyRequest()
        .authenticated();
    }
}
```

授权服务使用用户信息的查询接口，所以也是一个资源服务，在这里拦截了所有的链接地址。

④ 新建 SecurityConfiguration 进行安全配置。

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
        .inMemoryAuthentication()
        .withUser("book")
        .password("123456")
        .roles("USER");
    }

  @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
```

```
        }

    }
```

AuthenticationManagerBuilder

使用 inMemoryAuthentication() 在内存中配置用户信息，在指定用户角色时 roles() 会根据格式自动添加 ROLE_ 前缀。

AuthenticationManager

开启 password 授权模式。

⑤ 编写获取用户信息的 Controller。

```
@GetMapping("/user")
public Principal user(Principal user) {
  return user;
}
```

在访问各受保护的微服务模块时，各微服务模块需要向授权中心获取用户信息以完成对 access_token 的验证。

⑥ 在 appaction.properties 文件中配置。

```
spring.application.name=AUTH
server.port=${PORT:${SERVER_PORT:0}}
security.oauth2.resource.filter-order=3
eureka.client.service-url.defaultZone=http://localhost:8082/eureka
```

13.4.2　服务模块配置

服务调用者可能是通过网关服务发起的第三方应用，或者是模块之间的互相调用。每一次的调用都涉及到权限验证，在此可以新建两个 eureka 服务模块并使用 feign 调用方式模拟授权的过程。

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Server 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-api
version: 0.0.1-SNAPSHOT
packaging: jar
```

pom依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

② 新建 ResourceServerConfiguration 配置资源服务。

```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter{

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
        .csrf().disable()
        .authorizeRequests()
        .anyRequest()
        .authenticated()
          .and()
        .httpBasic();
    }

}
```

③ 新建 FeignClientConfiguration 配置 Feign。

```
@Configuration
public class FeignClientConfiguration {

    @Value("${security.oauth2.client.user-authorization-uri}")
    private String authorizeUrl;

    @Value("${security.oauth2.client.access-token-uri}")
    private String tokenUrl;

    @Value("${security.oauth2.client.client-id}")
    private String clientId;

    @Bean
    public RequestInterceptor oauth2FeignRequestInterceptor(OAuth2ClientContext oauth2Client
Context){
        return new OAuth2FeignRequestInterceptor(oauth2ClientContext, resource());
    }

    @Bean
    protected OAuth2ProtectedResourceDetails resource() {
        AuthorizationCodeResourceDetails resource = new AuthorizationCodeResourceDetails();
        resource.setAccessTokenUri(tokenUrl);
        resource.setUserAuthorizationUri(authorizeUrl);
        resource.setClientId(clientId);
        return resource;
    }

}
```

为安全考虑，各个服务之间使用 Feign 调用时也需要权限验证，通过 OAuth2FeignRequest Interceptor 在客户端授权的上下文中加入授权所需信息。

@Value 注解可以从 appaction.properties 文件中获取具体的配置信息。

- authorizeUrl：用户跳转去获取 access token 的 URI。
- tokenUrl：指定获取 access token 的 URI。
- clientId：客户端 id。

这里使用在授权中心定义的名为 client 的客户端 ID，由于是简单模式，并未设定客户端密码。

④ 在 application.properties 中配置。

```
spring.application.name=API
server.port=${PORT:${SERVER_PORT:0}}
security.oauth2.resource.id=api
security.oauth2.resource.user-info-uri=http://localhost:8080/uaa/user
security.oauth2.resource.prefer-token-info=false
security.oauth2.client.access-token-uri=http://localhost:8080/oauth/token
security.oauth2.client.user-authorization-uri=http://localhost:8080/oauth/
authorize
security.oauth2.client.client-id=client
eureka.client.service-url.defaultZone=http://localhost:8082/eureka/
```

security.oauth2.resource.id

指定资源的唯一标识。

security.oauth2.resource.user-info-uri

指定获取用户信息的地址，即之前在授权中心返回了 Principal 的 Controller。

security.oauth2.resource.prefer-token-info

是否检查 token 信息，设置为 false 时，则指定使用用户信息进行验证。

⑤ 编写 Controller。

```
@RequestMapping(value = "/private/{id}")
public String api(@PathVariable String id) {
  return "success : " + id;
}
```

这里只模拟了一个简单的 controller 以供测试，为了更为真实的环境，可以使用同样的配置再建立一个服务模块，使用 Feign 互相调用。

13.4.3 网关配置

① 新建 Spring Boot 工程，并在创建时勾选 Eureka Server 依赖。

Maven参数

```
groupId: org.book.rpc.cloud
artifactId: cloud-getway
```

```
version: 0.0.1-SNAPSHOT
packaging: jar
```

pom依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

② 新建 SecurityConfiguration 进行安全配置。

```
@Configuration
@EnableOAuth2Sso
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();
    }

}
```

使用 @EnableOAuth2Sso 注解开启 oauth 的单点登录模式，以免去进行用户校验时的频繁请求。

单点登录：Single Sign On，简称 SSO。SSO 使得在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

③ 在 application.properties 文件中配置。

```
spring.application.name=GETWAY
server.port=8080

zuul.routes.API.path=/api/**
zuul.routes.API.service-id=API
zuul.routes.API.sensitive-headers=
zuul.routes.AUTH.path=/uaa/**
zuul.routes.AUTH.service-id=AUTH
zuul.routes.AUTH.sensitive-headers=
zuul.add-proxy-headers=true

security.oauth2.client.access-token-uri=http://localhost:8080/uaa/oauth/token
security.oauth2.client.user-authorization-uri=http://localhost:8080/uaa/oauth/
authorize
security.oauth2.client.client-id=client
security.oauth2.resource.user-info-uri=http://localhost:8080/uaa/user
security.oauth2.resource.prefer-token-info=false
eureka.client.service-url.defaultZone=http://localhost:8082/eureka/
```

zuul.add-proxy-headers

是否在请求头部添加转发标识（X-Forwarded-*）。

13.4.4 测试运行

依次启动各工程后，便可在网关代理的请求中完成 OAuth2.0 验证。

获取token

请求方式：POST

请求地址：http://localhost:8080/uaa/oauth/token

URL 参数：用户名（username）、密码（password）、授权类型（grant_type）

Basic Auth 参数：客户端用户名（clientId）、客户端密码（Secret）

获取用户信息

请求方式：GET

请求地址：http://localhost:8080/uaa/user

URL 参数：access_token

请求服务

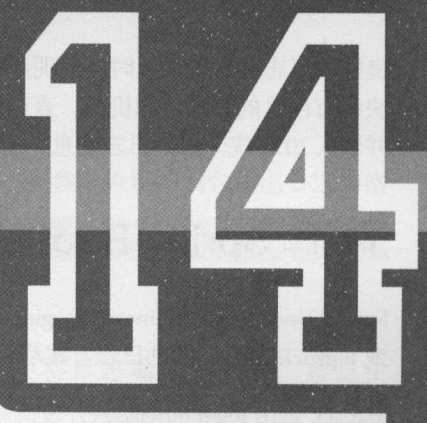
请求方式：GET

请求地址：http://localhost:8080/api/private/2

URL 参数：access_token

本章示例代码详见异步社区网站本书页面。

第14章　日志管理



14.1　Spring Boot 日志
14.2　分布式日志管理

良好的日志记录可以及时地发现系统中存在的问题，在分布式架构中由于有众多的服务模块，每个模块都有自己的日志管理机制，查询追踪的时候非常麻烦，为此可以结合 Spring Boot 的日志管理系统，并引入 ELK 统一收集日志信息，使整个日志分析工作变得简单。

14.1　Spring Boot 日志

Spring Boot 使用 commons-logging 和 slf4j 提供通用接口，日志具体实现则可由开发者自由选择 log4j 或 logback 方案。两种日志方案都可以通过配置使用控制台或者文件输出日志内容。

logback 是由 log4j 的创始人开发的新一代日志框架，用于替代 log4j。它效率更高、能够适应诸多的运行环境，也是 Spring Boot 推荐的日志实现方案。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

private Logger logger =  LoggerFactory.getLogger(this.getClass());

public void log() {
  logger.info("INFO级别信息");
  logger.debug("DEBUG级别信息");
  logger.error("ERROR级别信息");
  logger.warn("WARN级别信息");
  logger.trace("TRACE级别信息");
}
```

14.1.1　日志格式

Logging 输出的一条日志格式为：`%d %p ${PID} --- [%t] %c %msg %n`

```
    2017-08-27 17:31:53.972  INFO 15056 --- [restartedMain] org.apache.catalina.core.
StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.16
```

- `%d`：时间日期，精确到毫秒。
- `%p`：日志级别，ERROR、WARN、INFO、DEBUG 或者 TRACE。
- `${PID}`：进程 ID。
- `---`：分隔符，标识实际日志开始。
- `%t`：线程名，方括号括起来（可能会截断控制台输出）。
- `%c`：Logger 名，通常使用源代码的类名。

- %msg：日志内容。
- %n：换行。

在 appaction.properties 中通过 logging.pattern.console 参数设置控制台输出日志的格式，并通过 logging.pattern.file 参数设置输出到日志文件的格式。

14.1.2 输出到文件

默认情况下，Spring Boot 仅仅将日志输出在控制台。如果需要将日志输出到文件，需要在 application.properties 中配置 logging.file 或者 logging.path。

- logging.file：输出到指定的路径及文件名，可以为相对路径或者绝对路径。
- logging.path：输出到指定的路径，默认的文件名为 spring.log。

日志文件达到 10Mb 时，将会新开启一个文件输出，默认的日志输出级别为 ERROR、WARN 和 INFO，可以通过 logging.level.* 参数修改输出级别。

```
logging.level.root=WARN
logging.level.org.springframework=DEBUG
logging.level.org.hibernate=ERROR
```

logging.level.root

控制 root logger 的日志级别为 WARN 以上。

logging.level.org.springframework

控制 org.springframework 包下的日志级别。

logging.level.org.hibernate

控制 org.hibernate 包下的日志级别。

```
TRACE < DEBUG < INFO < WARN < ERROR < FATAL
```

日志等级是依次包含关系，设定了等级为 WARN，则会输出 WARN、ERROR、FATAL 级别的信息。

DEBUG 级别信息可以通过 debug=true 参数或在 Spring Boot 应用启动时，使用 java -jar target/appName.jar --debug 命令开启。

14.1.3 扩展配置

Spring Boot 约定了许多参数以减少开发中过多的配置量，对于日志系统也是如此，根据日志实现方案

只需创建不同的配置文件便可实现对日志的扩展。

- Logback：logback-spring.xml, logback-spring.groovy, logback.xml, logback.groovy
- Log4j：log4j-spring.properties, log4j-spring.xml, log4j.properties, log4j.xml
- Log4j2：log4j2-spring.xml, log4j2.xml
- JDK (Java Util Logging)：logging.properties

Spring Boot 官方推荐优先使用带有 -spring 的文件名作扩展的日志配置（如使用 logback-spring.xml，而不是 logback.xml）。

① 新建 logback-spring.xml 配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="60 seconds" debug="false">
    <property name="APP_NAME" value="LogDemo" />
    <contextName>${APP_NAME}</contextName>
    <include resource="org/springframework/boot/logging/logback/base.xml" />
    <jmxConfigurator />
    <logger name="org.springframework" level="INFO" />
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <File>${user.home}/logs/${APP_NAME}.log</File>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <FileNamePattern>${APP_NAME}.%d{yyyy-MM-dd}.log</FileNamePattern>
            <maxHistory>30</maxHistory>
        </rollingPolicy>
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>ERROR</level>
        </filter>
        <encoder>
            <Pattern>%d %p ${PID} --- [%t] %c %msg %n</Pattern>
        </encoder>
    </appender>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <Pattern>%d %p ${PID} --- [%t] %c %msg %n</Pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>
    <root level="DEBUG">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
```

```
    </root>
</configuration>
```

configuration

配置文件树形结构中的顶层标签。

- `scan`：当此属性设置为 true 时，配置文件如果发生改变，将会被重新加载。
- `scanPeriod`：设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，则默认间隔为 1 分钟。
- `debug`：当此属性设置为 true 时，将打印出 logback 内部日志信息，实时查看 logback 运行状态。

property

自定义参数。

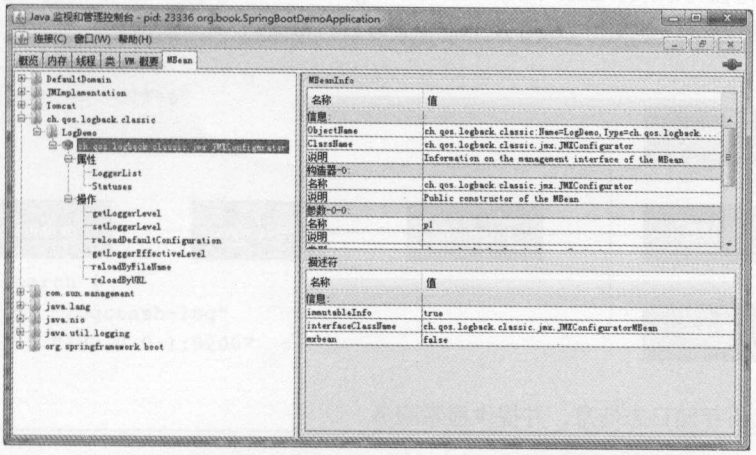
include

引入其他配置文件。

jmxConfigurator

开启 logback 的 JMX 支持。

② 通过命令 `jconsole` 打开 GUI 界面。



JMX（Java Management Extensions，即 Java 管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务管理应用。

logger

控制 org.springframework 包下的日志级别，与 `logging.level.*` 参数一致。

appender

配置输出日志的方案，可以有多个。

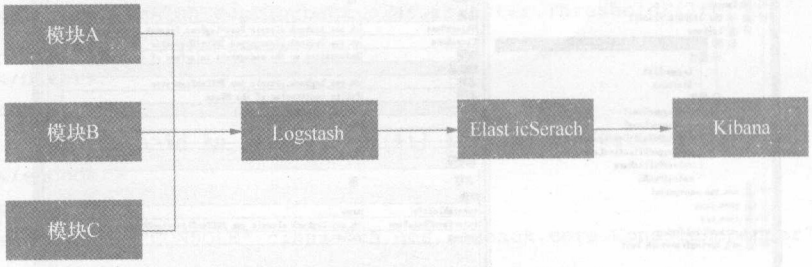
- `RollingFileAppender`：输出到文件。
- `rollingPolicy`：定义日志切分规则，通过 `TimeBasedRollingPolicy` 指定每天产生一个日志文件，并且通过 `maxHistory` 限定只保留最近 30 天的日志。
- `filter`：定义输出到日志的级别。
- `encoder`：定义日志格式。
- `ConsoleAppender`：输出到控制台。

root

必要节点，通过 `appender-ref` 使配置的 `appender` 生效。

14.2　分布式日志管理

分布式架构下的各个应用模块都记录着自身的日志，当这些应用部署在不同的机器时，想要排查程序的运行日志就变得极为麻烦。ELK（ElasticSerach Logstash Kibana）提供了统一的日志管理方案来解决这一问题，并且能够很好地支持 Log4j 及 Logback 等日志框架。



ElasticSerach：负责存储日志信息，并提供检索服务。

Logstash：负责收集应用发送的日志，并存入 ElasticSerach 中。

Kibana：提供一个可视化界面更为方便的分析日志。

14.2.1 ELK 搭建

ELK 之间存在一定的版本依赖关系，并且也建议将服务于业务及日志的 ElasticSearch 区分开来。所以下面的例子中将以 5.6.1 版本的 ELK 示例。

① 安装 ElasticSearch。

安装方式在 Spring Data Elasticsearch 中已经介绍，在此就不再赘述。与之不同的是这里使用 5.6.1 版本的 ElasticSearch。

使用默认配置，启动后在浏览器通过 http://localhost:9200 访问。

建立用于存储日志信息的索引 logstash-log，如下所示：

```
PUT http://localhost:9200/logstash-log
```

② 安装 Logstash。

下载地址：https://www.elastic.co/cn/downloads/logstash

下载解压后在 config 目录下新建 log_to_es.conf 配置文件，并写入以下配置信息：

```
input {
  tcp {
    port => 8082
    ssl_enable => false
    codec => json {
      charset => "UTF-8"
    }
  }
}

output {
  elasticsearch {
    index => "logstash-log"
    hosts => "127.0.0.1:9200"
  }
}
```

input

开启一个用于收集日志信息的 tcp 服务。

- port：自定义一个用于接收日志的端口号。

- ssl_enable：关闭 ssl。
- codec：基于 JSON 格式化日志信息，并指定编码格式为 utf-8。

output

将收集到的 TCP 服务发送到 elasticsearch 中。

- index：同于存储日志的索引名称。
- hosts：elasticsearch 所在地址。

配置完成后在 bin 目录中使用命令 `.\logstash -f ..\config\log_to_es.conf` 启动 logstash，并通过 `-f` 命令指定刚才编写的配置文件。

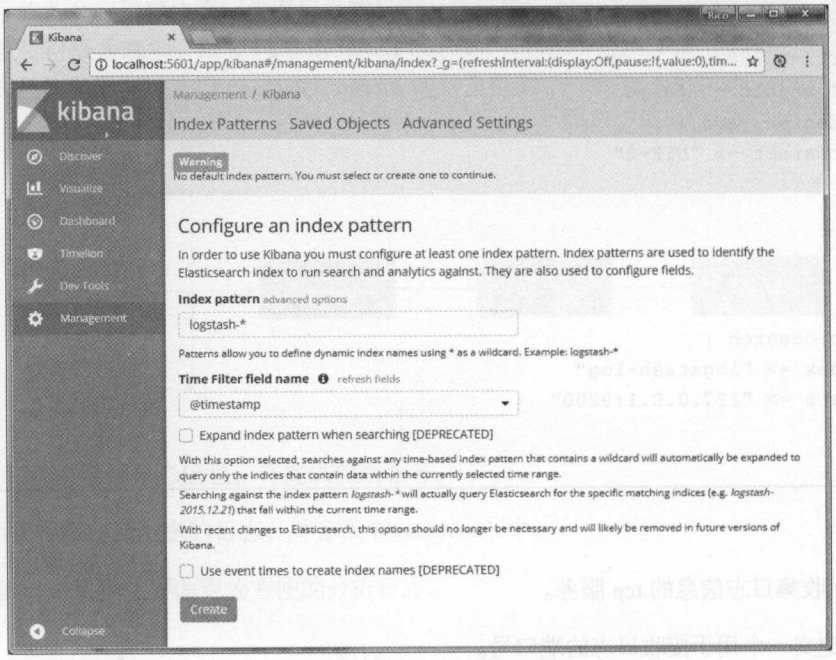
③ 安装 Kibana。

下载解压后在 config 目录下修改 `kibana.yml` 文件配置 kibana，指定 ElasticSearch 所在地址。

```
elasticsearch.url: "http://localhost:9200"
```

在 bin 目录运行 `.\kibana` 启动后，在浏览器访问 http://localhost:5601 便可看见管理界面。

在 kibana 管理界面中的 Management 目录下配置索引。



通过 index pattern 指定的 logstash-* 表示，kibana 将从 elasticsearch 中获取以 logstash 开头的所有索引。

14.2.2 Spring Boot 配置

① 在各微服务模块中的 pom.xml 文件中引入 logstash 依赖。

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.11</version>
</dependency>
```

② 新建 logback-spring.xml 配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
    <include resource="org/springframework/boot/logging/logback/base.xml" />

    <appender name="LOGSTASH" class="net.logstash.logback.appender.LogstashTcpSocket
Appender">
        <destination>localhost:8082</destination>
        <encoder charset="UTF-8" class="net.logstash.logback.encoder.LogstashEncoder" />
    </appender>

    <root level="INFO">
        <appender-ref ref="LOGSTASH" />
        <appender-ref ref="CONSOLE" />
    </root>

</configuration>
```

配置中使用 LogstashTcpSocketAppender 以 TCP 方式向 logstash 发送日志信息。

- destination：logstash 中用于接收日志的地址及端口号。
- encoder：指定日志编码格式。

③ 编写测试日志。

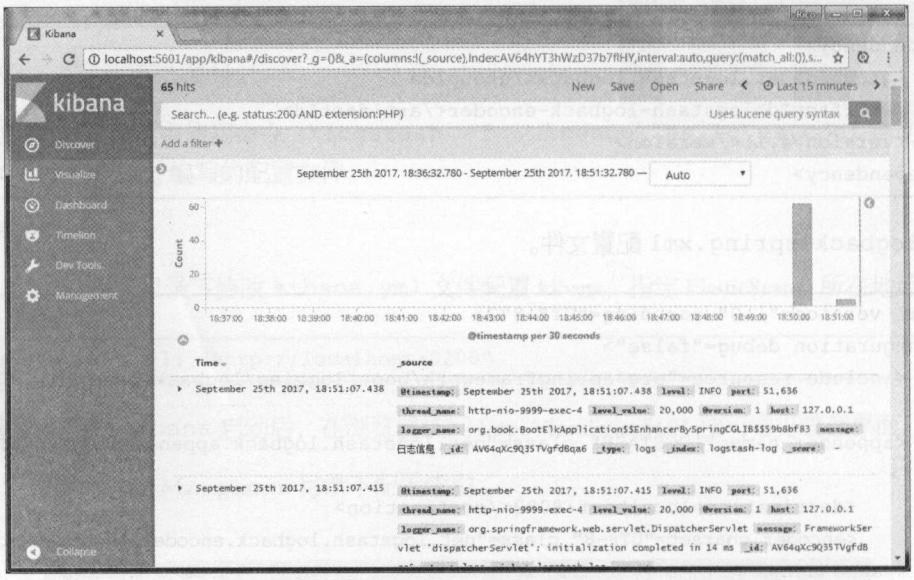
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public Logger logger = LoggerFactory.getLogger(this.getClass());
```

```
    logger.info("日志信息");
```

④ 测试运行。

启动 ELK 及应用后，使用浏览器访问 http://localhost:5601 便可在 discover 目录下看见日志信息。



第15章 热部署

由于Java静态语言的特性，在调试开发时需要不断地重启服务以运行最新的代码，这一过程机械而又繁琐。为了提高工作效率，Spring Boot基于Maven提供了spring-boot-devtools来监控应用中的各文件，当发生变动后自动触发重启应用。

① 在pom.xml中引入依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
</dependency>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <fork>true</fork>
      </configuration>
    </plugin>
  </plugins>
</build>
```

spring-boot-devtools

devtools核心类，由于Maven具有依赖集成关系，设置optional参数限定只在本应用执行热部署。

spring-boot-maven-plugin

Spring Boot的打包插件，用于生成可执行的jar，设置fork使devtools生效。

通过java -jar target/appName.jar运行Spring Boot应用时，devtools会认为是在生产环

境中运行，而不再触发重启机制。

② 在 Chrome 浏览器中安装 LiveReload 插件。

访问了一个请求后修改代码，触发了 devtools 重启应用，浏览器中的返回结果并未更新，需要重新发起请求以获得最新的结果。通过 Chrome 的 `LiveReload` 插件可省去重新发起请求这一步骤。

当以下目录中发生了变动时，应用不会重启，而是重新加载 devtools 的内置 `LiveReload` 服务。

- /META-INF/maven
- /META-INF/resources
- /resources
- /static
- /templates
- /public

③ 在 `appaction.properties` 中配置 devtools。

```
spring.devtools.restart.exclude=watch/**
spring.devtools.restart.additional-paths=/watch
spring.devtools.restart.enabled=false
```

spring.devtools.restart.exclude

排除监控的目录或文件，多个以逗号分隔。

`watch/**` 表示当前应用 resource 下的 watch 目录。

spring.devtools.restart.additional-paths

添加监控目录，多个以逗号分隔。

Windows 系统中，当前应用存放在 D 盘，`/watch` 则代表 D 盘下的 watch 目录。

spring.devtools.restart.enabled

是否打开重启监控。

第16章　接口文档管理

16

16.1　Dubbo 中使用 Swagger2
16.2　Spring Cloud 中使用 Swagger2

分布式架构中的各个服务最终产出 RESTful 风格的 API 接口，提供给前端（iOS、Android、Web）或其他第三方程序调用，一份丰富完整的接口描述文档能够大大降低沟通成本。本章将介绍如何使用 Swagger 轻松且方便地管理接口文档。

Swagger 由两部分组成。

- Swagger-Codegen：生成 JSON 格式的接口描述文档。
- Swagger-UI：提供界面解析接口描述文档。

Swagger 官网：https://swagger.io/

16.1 Dubbo中使用Swagger2

Swagger 主要在 Controller 中描述接口信息，在 Dubbo 架构中所有的服务都将由网关模块调用 Dubbo 服务转为 RESTful API，所以 Swagger 的整合也将在网关模块中进行。

① 在 pom.xml 文件中引入 Swagger2 依赖。

```
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-model</artifactId>
    <version>3.5.0</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
```

maven-model

用于解析 Maven 的 pom 文件中的内容。

springfox-swagger2

用于生成接口描述文档的核心依赖。

springfox-swagger-ui

用于解析接口描述的界面依赖。

② 新建 Swagger2Configuration 配置文件。

```
@Configuration
@EnableSwagger2
public class Swagger2Configuration {

    @Bean
    public Docket createRestApi() throws IOException, XmlPullParserException{
        MavenXpp3Reader reader = new MavenXpp3Reader();
        Model model = reader.read(new FileReader("pom.xml"));

        ApiInfo apiInfo = new ApiInfoBuilder()
                .title("接口名称")
                .description("接口介绍内容")
                .version(model.getVersion())
                .build();
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo)
                .select()
                .apis(RequestHandlerSelectors.basePackage("org.book"))
                .paths(PathSelectors.any())
                .build();
    }

}
```

MavenXpp3Reader

读取 Maven 的 pom 文件内容，并获取版本。

ApiInfo

用于配置该应用所产出的接口基本描述信息。

Docket

用于构建 Swagger 接口描述文档，并通过 API 指定接口所在的位置。

③ 新建用于接收数据的实体。

```
@ApiModel(description = "参数描述")
public class UserValidate {

    @ApiModelProperty(value = "用户昵称", required = true, example = "张三")
```

```
    private String nickname;

    @ApiModelProperty(value = "用户年龄", required = true, example = "20")
    private int age;

    // getter and setter
}
```

④ 新建用于返回数据的实体。

```
public class Customer {

    @ApiModelProperty(value = "用户昵称", example = "Rico")
    private String nickname;

    @ApiModelProperty(value = "用户年龄", example = "18")
    private int age;

    // getter and setter
}
```

⑤ 编写 Controller 描述接口。

```
@Api(description = "控制器名称")
@RestController
public class SwaggerController {

    @ApiOperation(value = "接口名称", notes = "使用URL传参方式描述接口")
    @ApiImplicitParams({
        @ApiImplicitParam(name = "age", value = "年龄", paramType = "path",
defaultValue = "2", dataType = "int"),
        @ApiImplicitParam(name = "nickname", value = "昵称", paramType = "query",
defaultValue = "张三", dataType = "string")
    })
    @ApiResponses({
        @ApiResponse(code = 200, message = "请求成功", response = Customer.class),
        @ApiResponse(code = 404, message = "资源未找到")
    })
    @RequestMapping(value = "api/{age}/{nickname}", method = RequestMethod.GET,)
    public Customer api(@PathVariable("age") int age, @RequestParam("nickname")
String nickname) {
        Customer customer = new Customer();
        customer.setAge(age);
        customer.setNickname(nickname);
        return customer;
```

```
    }

    @ApiOperation(value = "接口名称", notes = "使用实体传参方式描述接口", response =
Customer.class)
    @RequestMapping(value = "api", method = RequestMethod.POST)
    public Customer api(@RequestBody UserValidate userValidate) {
        Customer customer = new Customer();
        customer.setAge(userValidate.getAge());
        customer.setNickname(userValidate.getNickname());
        return customer;
    }

}
```

@Api

接口将以 Controller 进行分组，该注解用于描述该 Controller 信息。

@ApiOperation

描述该接口的信息。

- value：接口名称。
- notes：接口详细描述。

@ApiImplicitParam

描述该接口所接受的参数信息。

- name：参数名称。
- value：参数介绍。
- paramType：参数接收类型。
- defaultValue：swagger ui 中显示的默认值。
- dataType：参数类型。

@ApiResponse

描述该接口的返回内容。

- code：HTTP 返回状态。
- message：状态描述。
- response：返回值类型。

描述实体参数与返回实体信息

通过实体接收参数以及将接口返回内容通过实体序列化为 JSON 已经是常用做法，如第 3 与第 4 步骤中所建立的实体所示，Swagger 可以通过 @ApiModelProperty 注解对实体中的参数进行描述。

- value：参数名称。
- example：默认数据。
- required：请求时该参数是否为必须的。

⑥ 测试。

完成了上述配置后，启动应用便可以通过 http://localhost:8080/v2/api-docs 访问到 Swagger 所生成的 JSON 格式 API 文档。

并通过 http://localhost:8080/swagger-ui.html 访问到 Swagger 的界面。并且展开接口后可以通过“Try it out!”按钮对该接口发起一次请求。

本小节示例代码详见异步社区网站本书页面。

16.2　Spring Cloud中使用Swagger2

在 Dubbo 中所有的 Controller 都集中在网关模块中，而 Spring Cloud 的各 Controller 分布在各微服务模块中，并由 Zuul 代理对外暴露。在配置 Swagger 也同样如此，每个微服务将产出各自的 JSON 格式

的 API 文档，并在 Zuul 网关中通过 Swagger UI 统一管理。

使用 OAuth2.0 对 API 接口进行安全管理是一个比较推荐的做法，所以在此示例将基于 13.4 小节增加 Swagger2 的配置。

16.2.1 微服务模块配置

① 在 pom.xml 文件中增加 Swagger2 依赖。

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>

<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-model</artifactId>
    <version>3.5.0</version>
</dependency>
```

② 新建 Swagger2Configuration 配置文件。

```
@Configuration
@EnableSwagger2
public class Swagger2Configuration {

    @Bean
    public Docket createRestApi() throws IOException, XmlPullParserException {
        MavenXpp3Reader reader = new MavenXpp3Reader();
        Model model = reader.read(new FileReader("pom.xml"));

        ApiInfo apiInfo = new ApiInfoBuilder()
                .title("接口名称")
                .description("接口介绍内容")
                .version(model.getVersion())
                .build();

        List<ResponseMessage> responseMessages = new ArrayList<>();
        responseMessages.add(new ResponseMessageBuilder().code(404).message("找不到资源").
build());
        responseMessages.add(new ResponseMessageBuilder().code(500).message("服务器
内部错误").build());
```

```
        return new Docket(DocumentationType.SWAGGER_2).apiInfo(apiInfo)
                .globalResponseMessage(RequestMethod.GET, responseMessages)
                .globalResponseMessage(RequestMethod.POST, responseMessages)
                .globalResponseMessage(RequestMethod.PUT, responseMessages)
                .globalResponseMessage(RequestMethod.DELETE, responseMessages)
                .select()
                .apis(RequestHandlerSelectors.basePackage("org.book"))
                .paths(PathSelectors.any())
                .build();
    }

}
```

Swagger2Configuration 的配置与之前在 Dubbo 中的配置基本一致，唯一不同的是 globalResponseMessage 的配置参数，在之前的配置中通过 @ApiResponse 注解来描述接口返回的状态所代表的意思，但是整个系统中所有的接口返回状态码都应该是一样的，这里为了避免重复设置 @ApiResponse 所以通过 globalResponseMessage 来进行统一处理

③ 在 ResourceServerConfiguration 文件中配置。

```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
        .csrf()
        .disable()
        .authorizeRequests()
        .antMatchers(HttpMethod.GET, "/v2/api-docs").permitAll()
        .and()
        .authorizeRequests()
        .anyRequest()
        .authenticated()
        .and()
        .httpBasic();
    }
}
```

将该模块生成的 JSON 格式 API 文档访问地址 /v2/api-docs 放开，不受 OAuth2.0 权限管理。

④ 编写 Controller。

```
@Api(value = "控制器名称", description = "控制器介绍")
@RestController
```

```
public class SwaggerController {

    @ApiOperation(value = "接口名称", notes = "接口介绍")
    @ApiImplicitParam(name = "id", value = "参数介绍", paramType = "path",
defaultValue = "2")
    @RequestMapping(value = "private/{id}",method = RequestMethod.GET)
    public String api(@PathVariable("id") int id) {
        return "success : " + id;
    }

}
```

16.2.2 网关模块配置

① 在 pom.xml 文件中加入依赖。

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.7.0</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.7.0</version>
</dependency>
```

② 在 SecurityConfiguration 文件中配置。

```
@Configuration
@EnableOAuth2Sso
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        .csrf()
        .disable()
        .authorizeRequests()
        .antMatchers("v2/api-docs/**", "/swagger-ui.html", "/webjars/**",
"favicon.ico", "/swagger-resource/**")
        .permitAll();
    }
}
```

将 Swagger UI 中相关资源放开，不纳入权限管理系统中。

③ 新建 Swagger2Configuration 配置文件。

```
@Primary
@Configuration
@EnableSwagger2
public class Swagger2Configuration implements SwaggerResourcesProvider {

    @Override
    public List<SwaggerResource> get() {
        return Arrays.asList(swaggerResource("api-server", "/api/v2/api-docs", "2.0"));
    }

    private SwaggerResource swaggerResource(String name, String location, String version) {
        SwaggerResource swaggerResource = new SwaggerResource();
        swaggerResource.setName(name);
        swaggerResource.setLocation(location);
        swaggerResource.setSwaggerVersion(version);
        return swaggerResource;
    }

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build();
    }

    @Bean
    public SecurityConfiguration securityInfo() {
        return new SecurityConfiguration(null, null, null, null, "Bearer access_token
具体内容", ApiKeyVehicle.HEADER, "Authorization", ",");
    }

}
```

SwaggerResourcesProvider

通过重写 get() 方法，指定 Swagger-UI 加载的 JSON 格式接口文档的地址。Swagger 会通过 Zuul 配置的路由规则自动寻找各指定的微服务接口文档链接地址。

SecurityConfiguration

当在 Swagger-UI 的接口中点击“try it out!”按钮发起请求时，由于接口被 OAuth2.0 协议管理起来，将会得到资源被保护的错误信息。通过注入 springfox.documentation.swagger.web.SecurityConfiguration 将 Access Token 加入请求头中已完成身份验证。

请求头示例

Authorization: Bearer 0b603a3e-6ff7-4a2f-bc80-29dd309e0524

④ 进行测试。

到此为止已经完成了 Spring Cloud 的 Swagger2 集成。通过 http://localhost:8080/swagger-ui.html 访问网关中的 Swagger-UI。通过 http://localhost:8080/api/v2/api-docs 访问服务模块的 api 接口描述文档。

本节代码详见异步社区网站本书页面。

第17章 Nexus私库

17

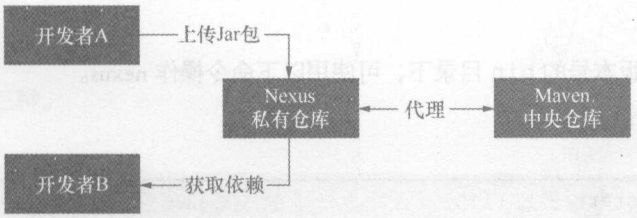
17.1 Nexus 安装

17.2 从 Nexus 私库下载 jar 包

17.3 上传 jar 包到 Nexus 私库

之前的示例都是以 maven 构建应用，只需在 pom.xml 文件中添加依赖信息，maven 便会自动从中央仓库中下载 jar 包，整个使用过程极其方便。在团队工作中通常会各自负责不同的业务模块，由于在不同的开发环境中工作，一些公共的 jar 包共享便成了问题，尤其是基于 Dubbo 框架开发时，服务调用方与提供方都要依赖独立出来的接口应用。

Nexus 提供了一个私有仓库，允许大家上传 jar 包，并且代理了 maven 的中央仓库，在获取依赖时对 pom.xml 文件的操作与之前并无区别。



17.1 Nexus 安装

选择对应的操作系统下载后解压会得到如下两个目录。

- nexus- 版本号：服务主目录，包含运行所需要的文件，如启动脚本、依赖 jar 包等。
- sonatype-work：仓库目录，包含生成的配置文件、日志文件、仓库文件等。

Windows版本

使用管理员身份运行 cmd 命令行工具，进入 nexus- 版本号的 bin 目录，可使用以下命令操作 nexus。

将 nexus 添加到 windows 服务中，如果不指定服务名，则默认为 nexus。

```
.\nexus.exe /install 服务名称
```

卸载服务。

```
.\nexus.exe /uninstall
```

启动 nexus 服务。

```
.\nexus.exe /start
```

关闭 nexus 服务。

```
.\nexus.exe /stop
```

在不加入 Windows 服务的情况下，前台启动 nexus。

```
.\nexus.exe /run
```

Linux版本

解压后进入 nexus-版本号的 bin 目录下，可使用以下命令操作 nexus。

后台启动 nexus。

```
sudo sh nexus start
```

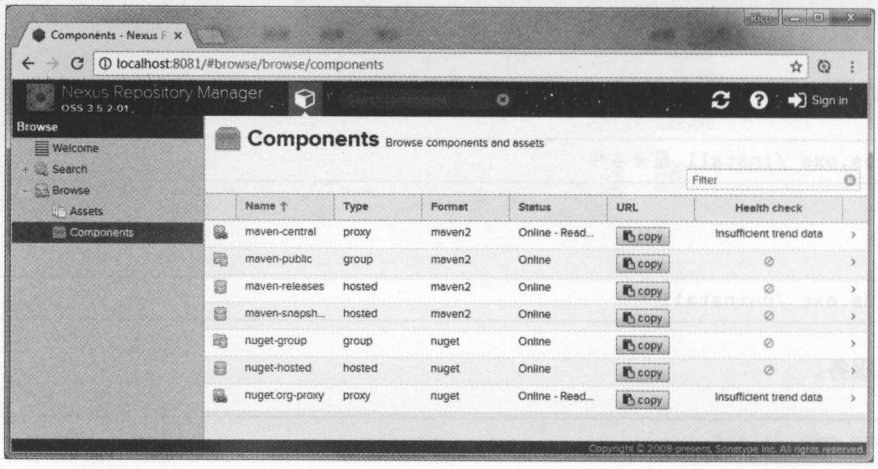
关闭 nexus。

```
sudo sh nexus stop
```

前台启动 nexus。

```
sudo sh nexus run
```

启动 nexus 服务后，在浏览器访问 http://localhost:8081，默认管理员账户为：admin，密码为：admin123。



maven-central

代理中央仓库。

maven-releases

存储 releases 版本的库。

maven-snapshot

存储 snapshot 版本的库。

maven-public

仓库集合，包含上面三种。

17.2 从Nexus私库下载jar包

单应用配置

在 pom.xml 文件中指定 nexus 地址。

```
<project>
    <repositories>
        <repository>
            <id>nexus</id>
            <name>nexus</name>
            <url>http://localhost:8081/repository/maven-public/</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>
</project>
```

- url：仓库地址，在 nexus 的仓库列表中，可以点击 copy 按钮获取。
- releases：允许下载 releases 版本的 jar 包。
- snapshots：允许下载 snapshots 版本的 jar 包，默认为关闭。

全局配置

修改 maven 的 config 目录中的 settings.xml 配置文件，添加 nexus 私库地址。

如果是通过 yum 或 apt 安装的 maven，可以先使用 whereis maven 命令查找 maven 地址。

```
whereis maven
maven: /etc/maven /usr/share/maven
```

settings.xml 文件位于 /etc/maven 目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                        http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <profiles>
        <profile>
            <id>nexusProfile</id>
            <repositories>
                <repository>
                    <id>nexus</id>
                    <name>nexus Repository</name>
                    <url>http://localhost:8081/repository/maven-public</url>
                    <layout>defult</layout>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                </repository>
            </repositories>
        </profile>
    </profiles>
    <activeProfiles>
        <activeProfile>nexusProfile</activeProfile>
    </activeProfiles>
</settings>
```

指定开发工具对 maven 的配置文件。

Eclipse：window → preferences → maven → user settings

打开 User Settings 界面后，在 user settings 或 global settings 中指定 setting.xml 配置文件。

IntelliJ IDEA：file → settings → Build,Execution,Deployment → maven

打开 maven 配置界面后，在 user settings 中指定 settings.xml 配置文件。

17.3 上传jar包到Nexus私库

① 在 settings.xml 文件中添加 nexus 私库权限。

```
<servers>
    <server>
        <id>nexus</id>
        <username>admin</username>
        <password>admin123</password>
    </server>
</servers>
```

② 在 pom.xml 文件中指定上传的仓库。

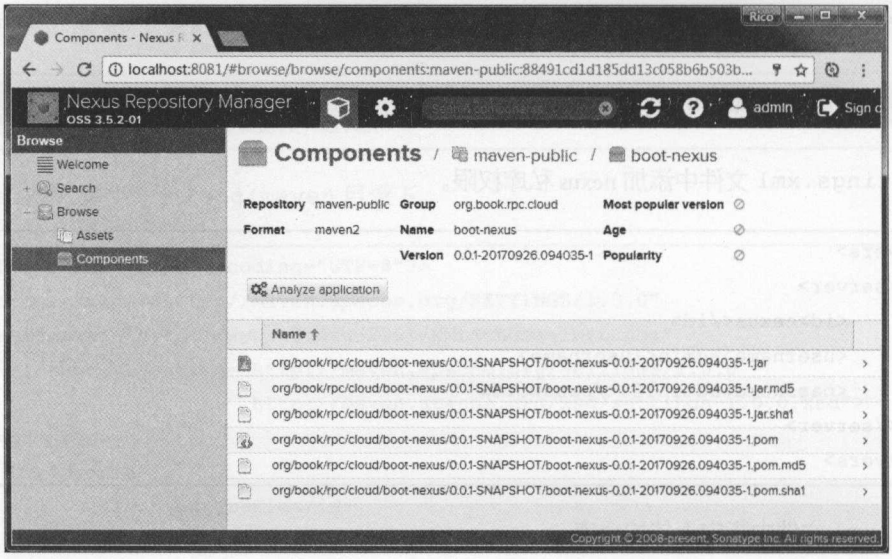
```
<distributionManagement>
    <repository>
        <id>nexus</id>
        <name>nexus-releases</name>
        <url>http://localhost:8081/repository/maven-releases/</url>
    </repository>
    <snapshotRepository>
        <id>nexus</id>
        <name>nexus-snapshot</name>
        <url>http://localhost:8081/repository/maven-snapshots/</url>
    </snapshotRepository>
</distributionManagement>
```

- repository：指定 repository 版本的 jar 包上传地址。
- snapshotRepository：指定 snapshot 版本的 jar 包上传地址。
- id：需要与 settings.xml 文件中配置的 server 的 id 一致，用于关联上传时的权限验证。

③ 上传 jar 包。

使用命令行工具进入应用所在根目录，并通过 mvn deploy 命令上传 jar 包。

上传完成后便可在 nexus 中查看到已上传的 jar 包。



引用私库的 jar 包与之前并无区别，只需在 pom.xml 文件中指定相应的 groupId、artifactId、version 信息便可。

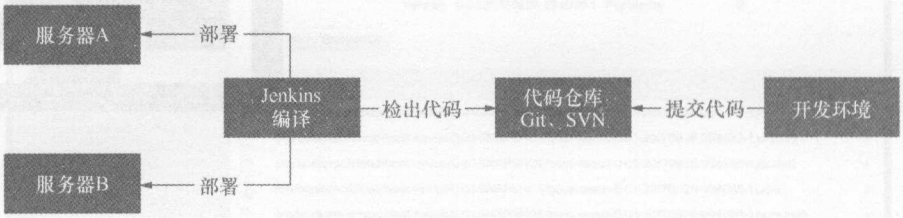
```
<dependency>
    <groupId>org.book.rpc.cloud</groupId>
    <artifactId>boot-nexus</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

第18章　发布系统

18.1　Jenkins 安装配置

18.2　构建任务

与单应用相比分布式架构会编译出多个 jAR 包，在快速版本迭代的开发环境中，需要频繁地将这些 jar 包部署到多台服务器上，这显然是一件枯燥且繁琐的事情。Jenkins 可以很好地解决这一问题，它可以自动将代码从 Git 或 SVN 仓库中检出并编译，最后将编好的 jar 包发送到目标服务器（Linux）中完成部署，同时还可以指定触发编译规则与记录通知结果，使整个构建过程全自动化，最终实现系统持续集成。



> 持续集成：Continuous Integration（CI）是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽快地发现集成错误。许多团队发现这个过程可以大大减少集成的问题，让团队能够更快地开发内聚的软件。

18.1　Jenkins 安装配置

根据操作系统选择 Jenkins 版本或者直接选择 War 包部署在 Tomcat 中。完成部署后便可通过浏览器访问 http://localhost:8080 完成安装。

- 第一步：提交指定路径文件中的密码完成权限验证。
- 第二步：安装插件。
- 第三步：设定管理员账户密码。

安装插件

Spring Boot 基于 Maven 进行构建，并最终将编译好的 jar 包发送至 Linux 服务器中进行部署。

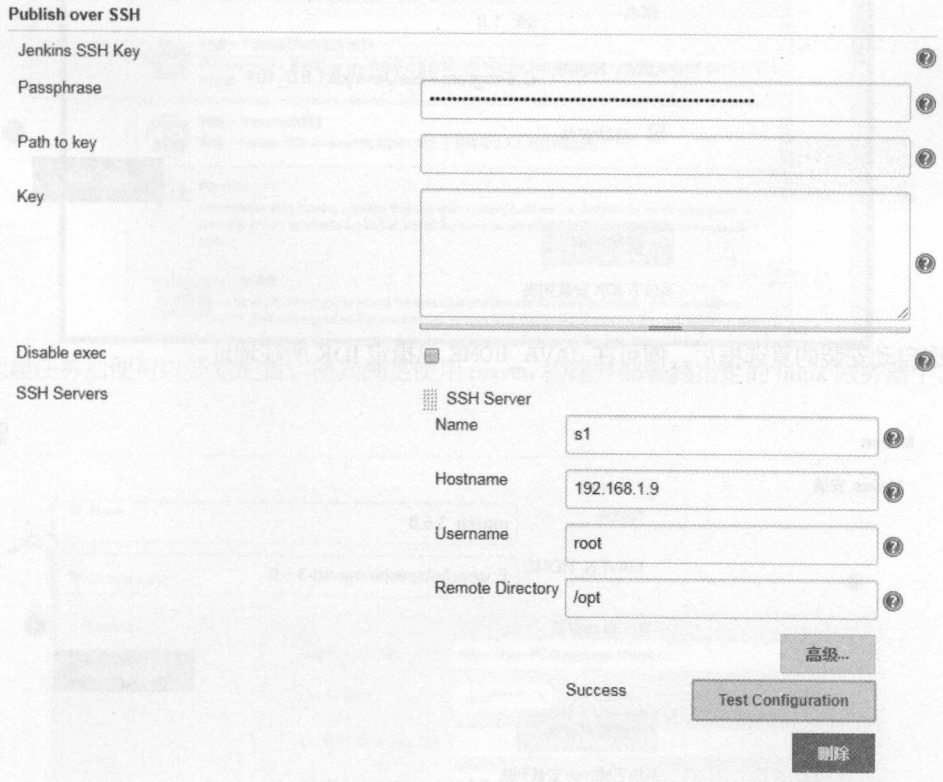
所以需要在 Jenkins → 系统管理 → 插件管理 中安装以下两个插件。

- Maven Integration plugin

- Publish Over SSH

配置部署服务器

安装完 Publish Over SSH 后便可在 Jenkins → 系统设置 → Publish over SSH 中配置用于部署应用的服务器账号密码相关信息。

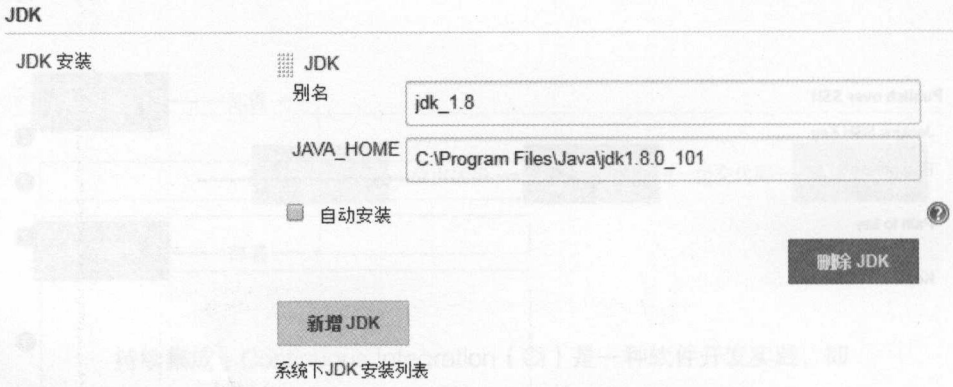


- Passphrase：密码
- Name：名称
- Hostname：访问地址
- Username：用户名
- Remote Directory：指定上传编号的 jar 包所在地址

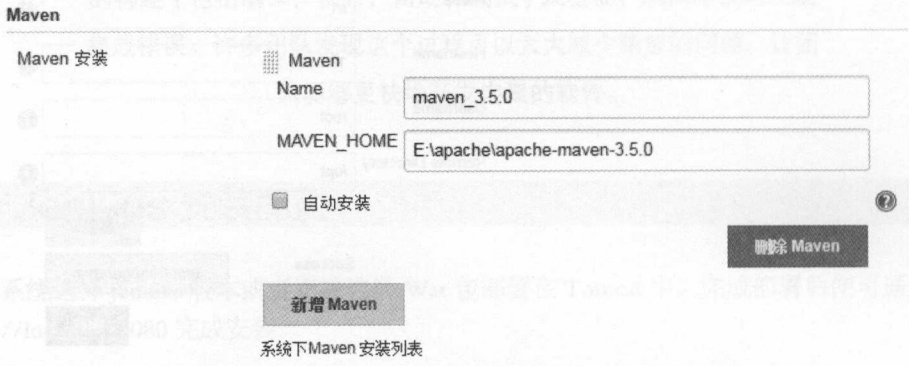
完成配置后点击 Test Configuartion 按钮测试，返回 Success 时表示服务器连接成功。

配置编译工具

编译的过程由 Jenkins 管理，所以需要在 Jenkins → 系统管理 → Global Tool Configuration 中指定 JDK 与 Maven。



取消勾选自动安装的复选框后，便可在 JAVA_HOME 中指定 JDK 所在地址。



与配置 JDK 一样，在 Maven 选项中通过 MAVEN_HOME 指定 Maven 所在地址。

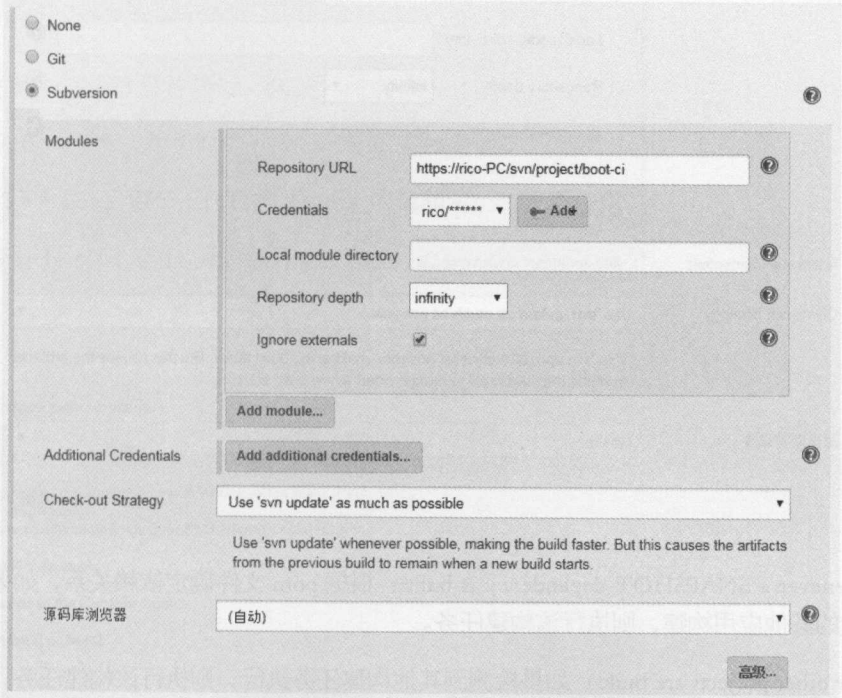
18.2　构建任务

在创建构建任务之前，需要先准备一个 Spring Boot 应用，并提交到 SVN 或 Git 中，当一切准备工作就绪后，便可在 Jenkins 中选择“构建一个 maven 项目”开始进行任务的配置工作。

“构建一个 maven 项目”的选项来自之前安装的 Maven Integration plugin 插件。

新建完构建任务后便可以开始配置，使其满足使用 maven 构建并部署到指定的 linux 服务器中。

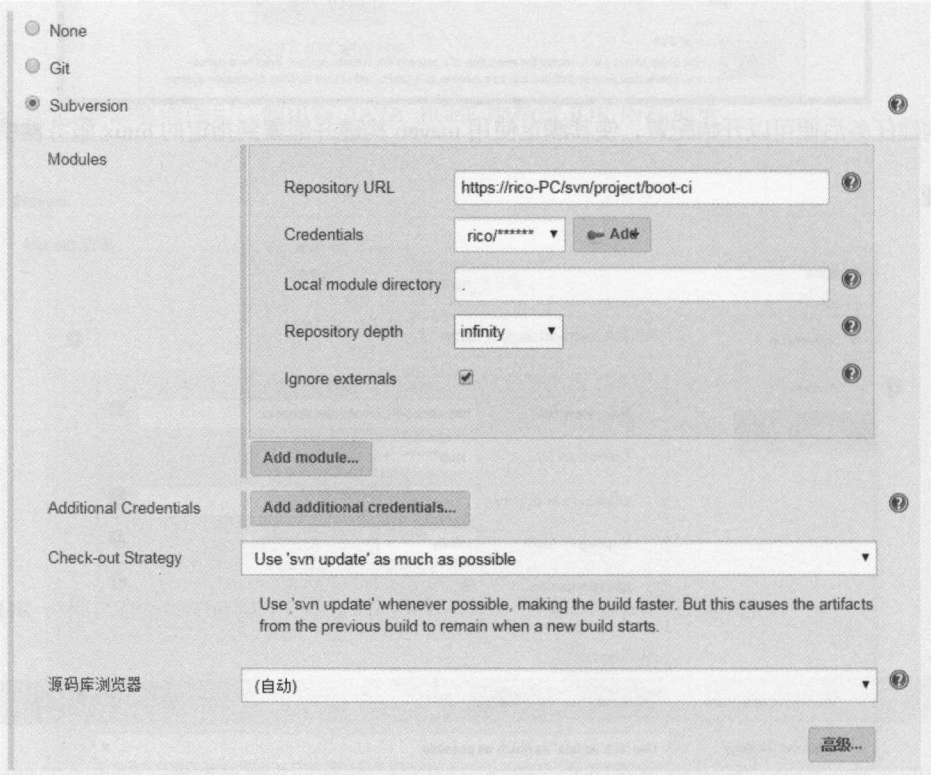
源码管理



Jenkins 会自动从代码仓库中检出代码，所以这里需要指定所使用的代码仓库及地址信息。

- Repository URL：仓库所在地址。
- Credentials：访问 SVN 所使用的权限配置信息，点击 add 按钮添加用户名与密码，并选择刚才所配置的即可。
- Local module directory：需要构建的项目名称，由于在仓库地址中直接指定到了项目名称，所以此处可不填。
- Repository depth：指定从仓库检出代码时的深度策略，infinity 表示所有文件及目录。
- Ignore externals：忽略外部环境。

构建触发器



- Build whenever a SNAPSHOT dependency is built：根据 pom 文件确定依赖关系，如果该构建任务中所依赖的其他应用构建，则执行该构建任务。
- Build after other projects are built：如果检测到其他构建任务执行，则执行该构建任务。

- Build periodically：根据指定的时间执行构建任务。
- Poll SCM：根据指定的时间点检查代码，如果代码被修改则执行构建任务。

Build periodically 与 Poll SCM 指定时间的表达式为：* * * * *

位置	描述	取值范围
1	分钟	0 至 59
2	小时	0 至 23
3	天	1 至 31
4	月	1 至 12
5	周	0 至 7，0 与 7 都代表星期天

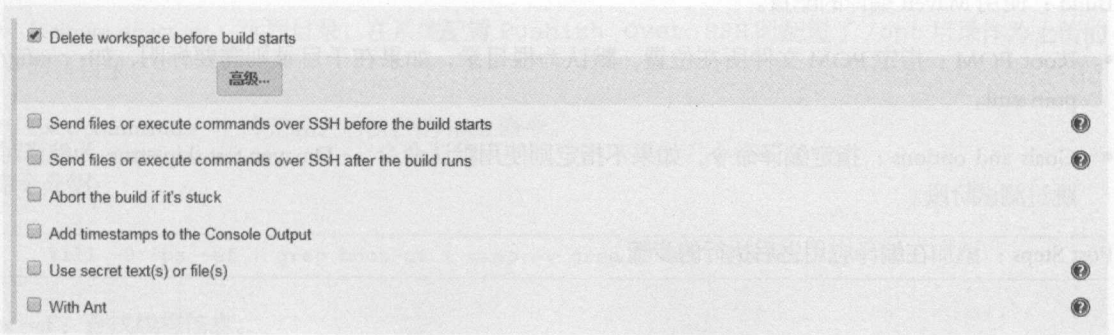
特殊格式。

- *：所有有效值。
- M-N：指定值的范围。
- M-N/X、*/X：以 X 为步长，指定的区域或整个有效范围区间的值。
- ,：分隔符，用于指定多个值。

示例。

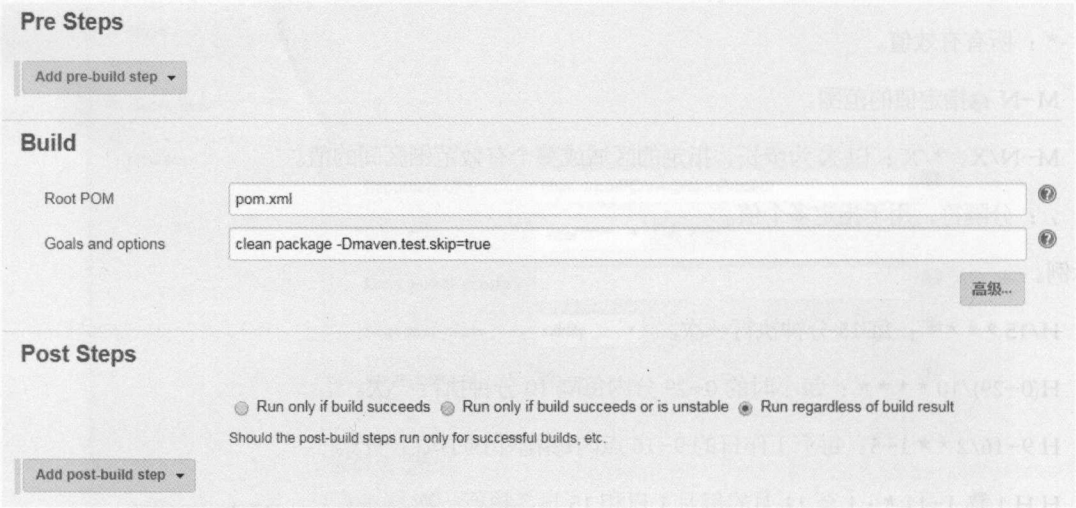
- H/15 * * * *：每 15 分钟执行一次。
- H(0-29)/10 * * * *：每小时的 0-29 分内每隔 10 分钟执行一次。
- H 9-16/2 * * 1-5：每个工作日的 9-16 点内每隔两小时执行一次。
- H H 1,15 1-11 *：1 至 11 月的每月 1 日和 15 日各执行一次。

构建环境



- Delete workspace before build starts：构建之前删除工作区。
- Send files or execute commands over SSH before the build starts：在代码检出工作区，并在构建运行之前向目标服务器发送文件或命令。
- Send files or execute commands over SSH after the build runs：在构建完成之后向目标服务器发送文件或命令。
- Abort the build if it's stuck：如果构建出现问题则终止。
- Add timestamps to the Console Output：控制台输出增加时间戳。
- Use secret text(s) or file(s)：使用加密文本或文件。

编译应用



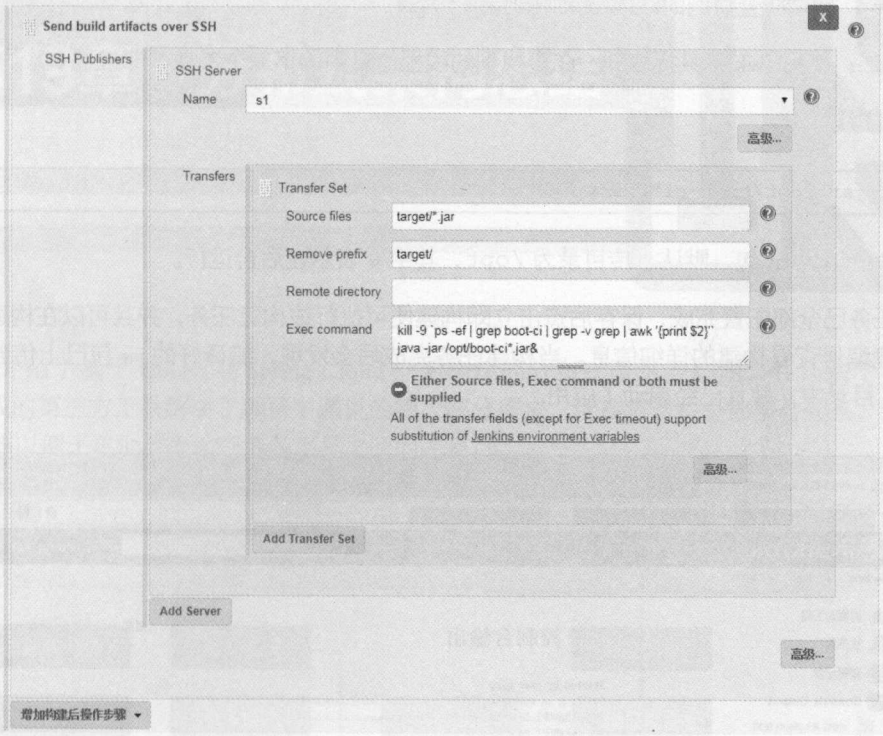
Pre Setps：添加编译应用之前执行的步骤。

Build：使用 Maven 编译的配置。

- Root POM：指定 POM 文件所在位置，默认为根目录，如果在子目录则需要标明，如：config/pom.xml。
- Goals and options：指定编译命令，如果不指定则使用默认命令，-Dmaven.test.skip=true 为编译时跳过测试阶段。

Post Steps：添加在编译应用之后执行的步骤。

构建后操作



添加构建完成后的操作，这里将构建后的 jar 包通过 Publish Over SSH 发送到目标服务器中，并通过 shell 命令关闭之前运行的应用，运行新编译后的应用以完成部署。

使用 Maven 编译应用后，会将编译结果存放在工程目录下的 /target/ 目录中，默认 jar 包名称格式为：artifactId-version.jar，例如 boot-ci-0.0.1-SNAPSHOT.jar。

- Source files：需要上传的文件。
- Remove prefix：移除目录。
- Remote directory：远程目录，在系统配置 Pushish Over SSH 时配置了 /opt 目录作为上传的默认目录。
- Exec command：上传完成之后执行的 shell 命令。

命令介绍：

```
kill -9 `ps -ef | grep boot-ci | grep -v grep | awk '{print $2}'`
```

ps -ef：查找线程信息。

grep boot-ci：在查找到的线程结果中过滤包含 boot-ci 的线程信息。

grep -v grep：从过滤查找到的线程结果中排除来自 grep 命令的信息。

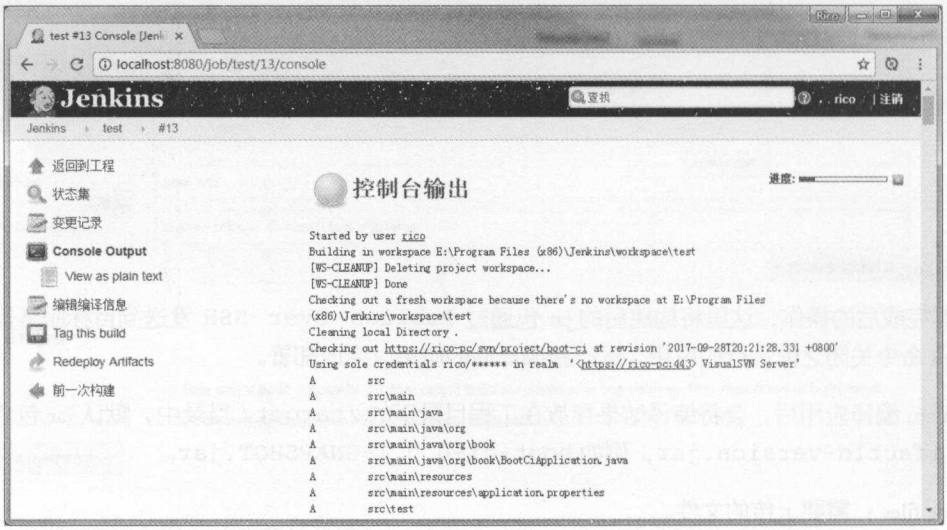
awk 'print $2'：使用 awk 工具提取 ps -ef 查找到的线程信息中的第二个参数，即线程 ID。

kill -9：关闭线程 ID。

```
java -jar /opt/boot-ci*.jar&
```

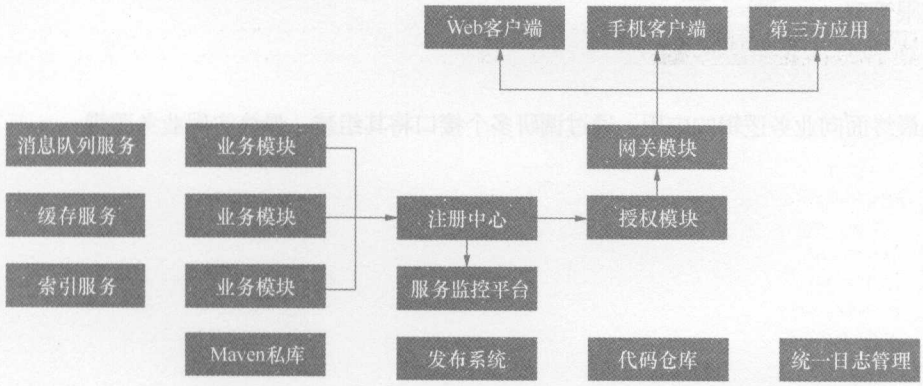
使用 java 命令运行 jar 包，默认上传目录为 /opt，其中 & 表示在后台运行。

至此构建任务已全部配置完成，保存后点击立即构建便可执行该构建任务，并且可以在构建历史列表中点击构建编号查看构建的详细信息。当构建任务完成后会发现，编译好的 jar 包已上传到目标服务器的 /opt 目录中，并且已经启动了应用。



第19章　分布式架构总结

目前已经介绍了基于 Dubbo 与 Spring Cloud 框架构建的分布式微服务应用，并且基于 Spring Boot Starter 集成的第三方工具解决了项目中遇见的绝大部分需求，现在将这些全部整合起来，绘制出完整的架构图谱以便于更加清晰地理解各工具之间的关系。



注册中心

无论是 Dubbo 的 Zookeeper 还是 Spring Cloud 的 Eureka 都负责所有服务模块的自动注册与发现，是整个分布式架构最为核心的部分，所以在生产环境中通常部署多个注册中心实现集群化以提高可用性。

业务模块

大部分的企业级应用都是围绕着数据库（MySQL）的 CURD 操作编写，当数据库达到 I/O 瓶颈后，一般会对其进行分库分表，使用主从库的方式部署在多台机器上以获得更大的计算量。分布式架构中的各业务模块天生就支持这一特性，各业务模块部署在不同的计算机上，只需让每个模块拥有各自独立的数据源便可。

分布式架构师基于网络进行远程调用，为了更高的可用性甚至可以将服务模块部署在异地的多个机房，当其中一个机器发生故障后依然可以由远程的机房中的应用实例提供服务，并且可以与 CDN 配合起来组建网络。

独立服务

消息队列（RabbitMQ）、缓存服务（Redis）、索引服务（ElasticSearch）并不直接参与到业务逻辑中，而是为各个独立业务模块提供公共服务，为了更好的稳定性也可以将各独立服务进行集群化部署。

服务监控平台

Dubbo 的 Mock 与 Spring Cloud 的熔断器都可以反映出服务之间的调用情况，并且各自的监控平台都能够反映出各业务模块的健康状况。它极大地减轻了运维的工作，在整个架构中属于必不可少的一块。

网关与授权模块

所有业务模块产出的服务接口由网关模块代理并向外提供服务，每次接口的请求都要经过网关模块，为了避免因请求量过大而造成网关模块崩溃，可以在 Dubbo 架构中为网关模块配置 Nginx 负载均衡，而 Spring Cloud 则可以直接在 eureka 中注册多个 zuul 网关以达到负载均衡的目的。

为了确保每个业务模块的独立性，开发人员更倾向于产出无状态的接口，而不是基于 Session 实现权限验证。所有的权限验证应交由基于 OAuth2.0 协议实现的授权模块，并配合网关模块统一对所有接口进行权限管理。

客户端

客户端是最终面向业务逻辑的应用，通过调研多个接口将其组装，最终实现业务逻辑。

读者好评

这是一本“手把手”技术分享书籍，内容非常实用并具可操作性。Rico就像一个技术世界的行者，和读者肩并肩漫游微服务的世界。相信这本书，能够让读者在保持技术新鲜度的学习过程中节约大量的时间。

陈雷
滴滴顺风车技术经理
高级专家工程师

本书涵盖了分布式微服务开发部署的方方面面，对于刚刚接触微服务架构而又无从下手的读者大有裨益。

李程
研发工程师

“微服务”这个主题不容易讲好，因为内容不仅涉及构建一个可靠系统的方方面面，还需要体现出其架构核心优势里的“微”字。然而作者深入浅出地把这个主题说得通俗易懂，这充分体现出作者深厚的技术底蕴。这本书是作者厚积薄发之作，涵盖了构建“微服务”架构的关键技术，兼具实用性和前瞻性。书中提供了很多短小精悍的代码片段，让你轻松搭建基于“微服务”的应用。

李晓超
大数据研发工程师

微服务架构是这几年技术发展趋势，很多公司都开始着手转型微服务。作者通过本书，就当前一些成熟的微服务解决方案都进行了详细的描述。通过具体的实战，带你覆盖微服务技术的各方面要点。值得一读。

徐鑫
微店研发工程师

本书介绍了使用主流框架快速构建微服务分布式架构系统，并以解决实际项目问题为目标，借助第三方服务所提供的解决方案，完善整个分布式系统。

本书的主要内容包括：

- 微服务架构基本概念
- 分布式架构基本模块拆分逻辑
- Spring Boot基础
- 使用Dubbo构建微服务分布式系统
- 使用Spring Cloud构建微服务分布式系统
- MySQL、MongoDB、ElasticSearch、Redis的集成与使用
- 使用Hibernate Validator验证接收的表单数据
- 使用Spring Task在单机应用中创建定时任务
- 在分布式环境中使用QuartZ创建定时任务
- 在分布式环境中使用Spring Session进行Session共享
- RabbitMQ消息队列的集成方式
- 基于Spring Boot建立Web应用
- 使用FreeMarker模板引擎渲染界面
- 统一异常处理方式
- 使用Spring Security实现OAuth2.0协议的权限管理解决方案
- Spring Boot日志管理
- ELK分布式环境中的日志收集解决方案
- Spring Devtools热部署
- 使用Swagger快速生成接口文档
- 使用Nexus解决分布式环境中JAR包共享方式
- 使用Jenkins自动编译及发布服务

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

ISBN 978-7-115-47558-9

ISBN 978-7-115-47558-9
定价：59.00 元

分类建议：计算机／系统架构
人民邮电出版社网址：www.ptpress.com.cn